Component-wise Incremental LTL Model Checking

Vince Molnár[1], András Vörös[1], Dániel Darvas[1], Tamás Bartha[2] and István Majzik[1]

[1]Department of Measurement and Information Systems, Budapest University of Technology and Economics, Hungary
[2]Institute for Computer Science and Control, Hungarian Academy of Sciences, Hungary

Abstract. Efficient symbolic and explicit-state model checking approaches have been developed for the verification of linear time temporal logic (LTL) properties. Several attempts have been made to combine the advantages of the various algorithms. Model checking LTL properties usually poses two challenges: one must compute the synchronous product of the state space and the automaton model of the desired property, then look for counterexamples that is reduced to finding strongly connected components (SCCs) in the state space of the product. In case of concurrent systems, where the phenomenon of state space explosion often prevents the successful verification, the so-called saturation algorithm has proved its efficiency in state space exploration. This paper proposes a new approach that leverages the saturation algorithm both as an iteration strategy constructing the product directly, as well as in a new fixed-point computation algorithm to find strongly connected components on-the-fly by incrementally processing the components of the model. Complementing the search for SCCs, explicit techniques and component-wise abstractions are used to prove the absence of counterexamples. The resulting on-the-fly, incremental LTL model checking algorithm proved to scale well with the size of models, as the evaluation on models of the Model Checking Contest suggests.

Keywords: symbolic model checking; LTL; saturation; component-wise abstraction; SCC computation; incremental algorithm

1. Introduction

Model checking is a formal verification technique to analyze properties of system designs. Given a specification and a model, the goal of model checking is to decide whether the model satisfies the specification or not. Linear temporal logic (LTL) specifications play an important role in the history of verification as being a prevalent formalism to specify the requirements of reactive and safety-critical systems [Pnu77]. The behaviors defined by an LTL property can be expressed with the help of a so-called *Büchi automaton* [Büc62]. The language of a Büchi automaton consisting of infinite words can characterize the behaviors violating the LTL

Correspondence and offprint requests to: Vince Molnár, Budapest University of Technology and Economics, Magyar tudósok körútja 2., H-1117 Budapest, Hungary. e-mail: molnarv@mit.bme.hu

Authors' manuscript. Published in *Formal Aspects of Computing* 28(3):345–379, 2016. © British Computer Society, 2015. The original publication is available at springerlink.com. DOI: 10.1007/s00165-015-0347-x.

specification. The problem of checking LTL properties is usually reduced to deciding language emptiness of the synchronous product of two Büchi automata: one characterizing the possible behaviors of the system and another accepting behaviors that violate the desired property [VW86]. The language emptiness problem of the result product Büchi automaton can be decided by finding *strongly connected components* (SCCs).

Explicit state algorithms exist for LTL model checking based on a step-by-step traversal of the state space (e. g., [HPY97]). However, the huge number of possible states of even simple systems can prevent these algorithms from succeeding. To overcome this weakness, special encoding of the states was proposed. This led to symbolic model checking algorithms which proved their efficiency in the last decades [McM92, BCM+92]. One of them is the so-called saturation algorithm which uses fine-grained decomposition and a special iteration strategy exploiting the component-wise structure of asynchronous systems [CMS03]. On the other side, these properties make the application of saturation in LTL model checking a complex task.

Abstraction is a key technique in the verification of complex systems, where the choice of the applied abstraction function determining the information to be hidden is very important. The computational cost of the abstraction is also significant: the more complex the chosen abstraction function is, the less efficient the model checking procedure might become. However, the computational investment of having better abstraction can pay off in decreasing the verification costs. Choosing the proper abstraction is difficult, many attempts exist targeting this problem, e. g., in [CGJ+00, HJMS02, WBH+06].

The goal of this paper is to combine the compact, component-wise state space representation of saturation with the efficiency of explicit state model checking. The first step is to utilize saturation to compute the synchronous product on the fly during the state space exploration. Then the model checking problem is split into smaller tasks according to the iteration of saturation and after each step an incremental model checking query is executed. An efficient, component-wise abstraction technique is used to construct small state graphs tractable by explicit-state algorithms.

Our contribution is a new hybrid LTL model checking algorithm that *1)* exploits saturation to build the symbolic state space representation of the synchronous product *2)* looks for SCCs on the fly, *3)* incrementally processes the discovered parts of the state space and *4)* uses explicit runs on multiple fine-grained abstractions to avoid unnecessary computations.

A new algorithm is introduced to build the product state space encoded by decision diagrams by using saturation. On-the-fly detection of SCCs is achieved by running searches over the discovered state space continuously during state space generation. In order to reduce the overhead of these searches, we present a new incremental fixed point algorithm that considers newly discovered parts of the state space when computing the SCCs. This approach relies on the component-wise structure of asynchronous systems and incremental fixed-point computation is driven by the ordering of the components. While this approach specializes on *finding* an SCC, a complementary algorithm maintains various abstractions of the state space to perform explicit searches in order to inductively prove the *absence* of SCCs.

High-level models like Petri nets or networks of automata express the structure of the system. This is exploited by saturation as it drives the traversal through the composite state space of the ordered components. Our approach further extends saturation in the direction of guiding model checking with the ordering of the components: incremental model checking is divided into smaller parts according to the component-wise structure of the system and model checking queries rely on the formerly computed results. In addition, component-wise abstraction prunes out unnecessary fixed point computations. Although example models are given as Petri nets, the algorithm can handle any discrete state model.

The paper is structured as follows. Section 2 presents the background of LTL model checking. An overview of the related work is in Section 2.5. Then saturation, a key algorithm for our work is introduced in Section 3. Then, the two main parts of our algorithm are introduced: Section 4 presents the symbolic synchronous product computation, Section 5 details the new symbolic SCC computation algorithm. The efficiency of SCC computation is further enhanced by various heuristics and abstractions, discussed in Section 6. The proposed new approach is evaluated and compared to three other tools in Section 7, then the work is concluded in Section 8.

This paper is an extended version of the conference paper [MDVB15]. While [MDVB15] focused on symbolic SCC detection, now the complete LTL model checking algorithm is introduced. Therefore we extended our previous paper with the saturation-based synchronous product automata computation and also the detailed description of the algorithms is included. It is also the first paper where the algorithms and their correctness are formalized and their proofs are introduced.

2. Background

This section overviews the background knowledge necessary for this work. Section 2.1 introduces the general concepts and the main formalisms of model checking. After that, Section 2.2 provides a short introduction to automata theory. Then Section 2.3 discusses the main challenges of LTL model checking, and Section 2.4 introduces the concepts of symbolic model checking.

2.1. Model Checking

Model checking is a formal verification method to verify finite state systems, i. e., to decide whether a given formal model satisfies a given requirement or not. The formal model and the requirement can be given with many different formalisms. Typically, the formal model is given by a Kripke structure or by a high-level model that can be transformed into a Kripke structure, such as Petri nets. The requirement is usually a temporal logic expression, i. e., a formula that expresses a temporal and logical statement. One of the most used temporal logic formalisms is LTL, a specification language using linear (non-branching) logical time. The rest of this section introduces Kripke structures, Petri nets and LTL in detail.

2.1.1. Kripke Structures

Kripke structures [Kri63] are directed graphs with labeled nodes. Nodes represent different states of the modeled system, while arcs denote state transitions. Each state is labeled with properties that hold in that state. This way, paths in the graph represent possible behaviors of the system. Labels along the paths give the opportunity to reason about sequences of states through their properties given as Boolean propositions.

Definition 1 (Kripke structure) Given a set of atomic propositions $AP = \{p, q, \ldots\}$, a (finite) Kripke structure is a 4-tuple $M = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, L \rangle$, where:

- $\mathcal{S} = \{s_1, \ldots, s_n\}$ is the (finite) set of states;
- $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states;
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation consisting of state pairs (s_i, s_j);
- $L : \mathcal{S} \to 2^{AP}$ is the labeling function that maps a set of atomic propositions to each state.

In the setting of LTL model checking, it is usually required that every state has at least one successor. This requirement is captured by defining the transition relation as left-total, i. e., for all $s_i \in \mathcal{S}$, there exists $s_j \in \mathcal{S}$ such that $(s_i, s_j) \in \mathcal{R}$. In that case, a *path* in M can be defined as an infinite sequence $\rho \in \mathcal{S}^\omega$ with $\rho(0) \in \mathcal{I}$ and $(\rho(i), \rho(i+1)) \in \mathcal{R}$ for every $i \geq 0$. The infinite sequence of sets of atomic propositions assigned to the states in ρ by L is called a *word* on the path and is denoted by $L(\rho) \in (2^{AP})^\omega$. The language described by a Kripke structure M (i. e., the set of all possible words on every path of M) is denoted by $\mathcal{L}(M)$.

2.1.2. Petri Nets

Petri net is a common modeling tool for formal verification with both graphical and mathematical representation. It is suitable to model concurrent and asynchronous systems with interleaving semantics.

Definition 2 (Petri net) An ordinary Petri net [Mur89] is a 5-tuple $PN = (P, T, E, w, M_0)$, where:

- $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places;
- $T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions ($P \cap T = \emptyset$);
- $E \subseteq (P \times T) \cup (T \times P)$ is a finite set of edges;
- $w \colon E \to \mathbb{Z}^+$ is the weight function assigning weights to edges;
- $M_0 \colon P \to \mathbb{N}$ is the initial token distribution (initial marking).

Graphically, a Petri net is a directed, weighted bigraph, where the two vertex sets are T (transitions) and P (places). Transitions are represented by rectangles, places are represented by circles. The tokens are shown as dots inside the places.

The *dynamic behavior* of the Petri net is determined by the *firing* of transitions. The rules of firing are the following:

- A transition $t \in T$ is *enabled*, iff for every place $p \in P$ where $(p,t) \in E$ (called *input* places of t) p is marked with at least as many tokens (denoted by $M(p)$) as the weight of the corresponding edge: $M(p) \geq w(p,t)$.
- The firing of an enabled transition t is nondeterministic.
- If the enabled transition t *fires*, it decreases the number of tokens for every input place p' by $w(p',t)$ and increases the number of tokens for every output place p'' by $w(t,p'')$.

2.1.3. Linear Temporal Logic

Linear temporal logic (LTL) [Pnu77] is a temporal logic with linear time model, meaning that it considers a single realized future behavior of a system, that is, a single path in a Kripke structure. The operators in LTL are the following: X, F, G, U and R.

An LTL formula φ is said to be *valid* with regard to a Kripke structure M, if it holds for all paths of M. It is *satisfiable* if it holds for some path in M. As a specification language, usually validity is desired.

Definition 3 (Syntax of LTL) The formal syntax of LTL is given by the following grammar in Backus–Naur Form (BNF), where $p \in AP$ is an atomic proposition:

$$\phi ::= \top \mid \bot \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \mathsf{X}\phi \mid \mathsf{F}\phi \mid \mathsf{G}\phi \mid [\phi\ \mathsf{U}\ \phi] \mid [\phi\ \mathsf{R}\ \phi]$$

A well-formed LTL formula is generated by ϕ.

Definition 4 (Semantics of LTL) The formal semantics of LTL is defined with respect to a Kripke structure M with a left-total transition relation (i. e., every path ρ of M is infinite). Let ρ be a path of M, and let $\rho[k]$ be a sub-path of ρ starting from element k. The relation $\rho \models \phi$ is again defined inductively:

1. $\rho \models \top$;
2. $\rho \models p$ iff $p \in L(\rho(0))$;
3. $\rho \models \neg\phi$ iff $\rho \not\models \phi$;
4. $\rho \models \phi_1 \wedge \phi_2$ iff $\rho \models \phi_1$ and $\rho \models \phi_2$;
5. $\rho \models \mathsf{X}\phi$ iff $\rho[1] \models \phi$;
6. $\rho \models [\phi_1\ \mathsf{U}\ \phi_2]$ iff for some $k \geq 0$, $\rho[k] \models \phi_2$ and for all $0 \leq i < k$, $\rho[i] \models \phi_1$.

The remaining temporal operators can be defined with the following equivalences:

1. $[\psi\ \mathsf{R}\ \neg\varphi] \equiv \neg[\neg\psi\ \mathsf{U}\ \neg\varphi]$
2. $\mathsf{F}\ \varphi \equiv [\top\ \mathsf{U}\ \varphi]$
3. $\mathsf{G}\ \varphi \equiv [\bot\ \mathsf{R}\ \varphi]$

As the definition of the semantics suggests, LTL can distinguish between words on paths of a Kripke structure, which are ω-regular words over 2^{AP} as discussed in Section 2.1.1. This way, it is possible to speak about such words satisfying an LTL formula φ, denoted by $w \models \varphi$, and the language of the formula $\mathcal{L}(\varphi) = \{w \mid w \models \varphi\}$, which is an ω-regular language.

2.2. Overview of Automata Theory

The automata-theoretical LTL model checking approaches, such as ours, heavily build on the theory of Büchi automata and especially their synchronous product, therefore they are introduced in detail in this section.

An *automaton* is an abstract machine that models some kind of computation over a sequence of input symbols. In formal language theory, automata are mainly used as a finite representation of infinite languages. In that context, they are often classified by the class of languages they are able to recognize.

The simplest class of automata is finite automata (also known as finite state machines). A finite automaton operates with a finite and constant amount of memory (independent of the length of the input). A finite automaton can operate on finite or infinite inputs. In formal language theory, inputs are called words, while elements of the input are called letters, the set of all possible letters constituting the alphabet.

A simple type of finite automata reading infinite words is the so-called Büchi automaton, which typically plays a key role in LTL model checking. This section overviews the definition of Büchi automata, the ways to represent other formalisms as an equivalent Büchi automaton and the synchronous product of such automata.

2.2.1. Büchi Automata

Büchi automaton [Büc62] is one of the simplest finite automata operating on infinite words.

Definition 5 (Büchi automaton) A Büchi automaton is a 5-tuple $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{I}, \Delta, \mathcal{F} \rangle$, where:

- $\Sigma = \{\alpha_1, \alpha_2, \ldots\}$ is the finite alphabet;
- $\mathcal{Q} = \{q_1, q_2, \ldots\}$ is the finite set of states;
- $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states;
- $\Delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is the transition relation consisting of state–input–state triples (q, α, q');
- $\mathcal{F} \subseteq \mathcal{Q}$ is the set of accepting states.

A *run* of a Büchi automaton *reading* an infinite word $w \in \Sigma^\omega$ is a sequence of automaton states $\rho_w \in \mathcal{Q}^\omega$, where $\rho_w(0) \in \mathcal{I}$ and $(\rho_w(i), w(i), \rho_w(i+1)) \in \Delta$ for all i, that is, the first state is an initial state and state changes are permitted by the transition relation. Let $inf(\rho_w)$ denote the set of states that appear infinitely often in ρ_w. The run ρ_w is called accepting iff it has at least one accepting state appearing infinitely often, i. e., $inf(\rho_w) \cap \mathcal{F} \neq \emptyset$. A Büchi automaton $\mathcal{A}$ accepts a word w iff it has an accepting run reading w. The set of all infinite words accepted by a Büchi automaton $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$ and is called the *language* of $\mathcal{A}$. The class of languages that can be characterized by Büchi automata is called ω-regular languages.

2.2.2. Kripke Structures and Büchi Automata

Although Kripke structures and Büchi automata are very similar in structure, Kripke structures are less expressive in terms of language, i. e., there are ω-regular languages that Kripke structures cannot produce. The following proposition presents a way to construct a Büchi automaton that accepts exactly the language of a Kripke structure.

Proposition 1 (Büchi automaton of Kripke structure) Given a Kripke structure $M = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, L \rangle$ with the set of atomic propositions AP, an equivalent Büchi automaton that accepts the language produced by M is $\mathcal{A}_M = \langle \Sigma, \mathcal{Q}, \mathcal{I}, \Delta, \mathcal{F} \rangle$, where:

- $\Sigma = 2^{AP}$, i. e., letter are sets of atomic propositions;
- $\mathcal{Q} = \mathcal{S} \cup \{init\}$, i. e., the states of the Kripke structure together with a special initial state;
- $\mathcal{I} = \{init\}$, i. e., the special initial state;
- $\Delta = \{(s, \alpha, s') \mid (s, s') \in \mathcal{R} \wedge \alpha = L(s')\} \cup \{(init, \alpha, s) \mid s \in \mathcal{I} \wedge \alpha = L(s)\}$, i. e., the automaton reads the labels of target states and additional transitions go from the special initial state to initial states of the Kripke structure[1];
- $\mathcal{F} = \mathcal{Q}$, i. e., every state is accepting.

Defining accepting states as the entire set of states is indeed necessary, because every path in a Kripke structure produces a word, no matter what states it passes. In other words, there is no such thing as *fairness* in ordinary Kripke structures. Based on that, an example ω-regular language that cannot be produced by a Kripke structure is a^*b^ω (infinitely many b's after finitely many a's). Any Büchi automaton accepting this language must have a loop that reads the letter a arbitrary many times and another loop that reads b infinitely many times. Only the second loop can contain an accepting state to force runs out of the first loop, which cannot be modeled in Kripke structures.

2.2.3. LTL to Büchi Automata

Since linear temporal logic formulae characterize a strict subset of ω-regular languages, there is an equivalent Büchi automaton for every LTL formula φ that accepts the same language that satisfies φ. In the history of model checking, many approaches have been developed to perform this conversion.

In general, the resulting automaton can be exponential in the size of the LTL formula, but efficient algorithms exist that are applicable in practice, where formulae are usually small. The algorithm described

[1] Technically, the special initial state is necessary to make the automaton read the labels of the initial states of the Kripke structure. The first step of the automaton can be regarded as the initialization of the corresponding Kripke structure.

in [GPVW95] was one of the first solutions, and there are some more advanced variants as well, such as [GO01].

Since atomic propositions in φ refer to labels of a Kripke structure, the alphabet of the equivalent automaton will be $\Sigma = 2^{AP}$. This is the same alphabet as that of the automaton describing the Kripke structure itself as defined in Proposition 1.

2.2.4. Synchronous Product of Büchi Automata

The synchronous product of two Büchi automata $\mathcal{A}_1$ and $\mathcal{A}_2$ over the same alphabet Σ is another Büchi automaton $\mathcal{A}_1 \cap \mathcal{A}_2$ that accepts exactly those words that both $\mathcal{A}_1$ and $\mathcal{A}_2$ accept [CGP99]. The language of the product automaton is therefore $\mathcal{L}(\mathcal{A}_1 \cap \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$. The construction of such synchronous product automata is well-known and can be found in e. g., [CGP99].

However, the setting of LTL model checking is special, as in one of the automata all states are accepting ($\mathcal{F}_1 = \mathcal{Q}_1$). In such a case, the definition of the product automaton is simpler. Since any infinite run in $\mathcal{A}_1$ will be accepting, the product inherits the accepting states of $\mathcal{A}_2$. This is the case when an automaton describes the possible behaviors of a Kripke structure (see Proposition 1).

Definition 6 (Synchronous product – special case) Given two Büchi automata $\mathcal{A}_1 = \langle \Sigma, \mathcal{Q}_1, \mathcal{I}_1, \Delta_1, \mathcal{F}_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, \mathcal{Q}_2, \mathcal{I}_2, \Delta_2, \mathcal{F}_2 \rangle$ with $\mathcal{F}_1 = \mathcal{Q}_1$, their synchronous product is $\mathcal{A}_1 \cap \mathcal{A}_2 = \langle \Sigma, \mathcal{Q}_\cap, \mathcal{I}_\cap, \Delta_\cap, \mathcal{F}_\cap \rangle$, where:

- $\mathcal{Q}_\cap = \mathcal{Q}_1 \times \mathcal{Q}_2$, i. e., product states are pairs;
- $\mathcal{I}_\cap = \mathcal{I}_1 \times \mathcal{I}_2$, i. e., every combination of the initial states will be considered;
- $\Delta_\cap = \{((q,r), \alpha, (q',r')) \mid (q, \alpha, q') \in \Delta_1 \wedge (r, \alpha, r') \in \Delta_2\}$, i. e., both automata can process the input;
- $\mathcal{F}_\cap = \mathcal{F}_1 \times \mathcal{F}_2 = \mathcal{Q}_1 \times \mathcal{F}_2$, i. e., accepting states of $\mathcal{A}_2$ are inherited.

The main challenge is the computation of $\Delta_\cap \subseteq (\mathcal{Q}_1 \times \mathcal{Q}_2) \times \Sigma \times (\mathcal{Q}_1 \times \mathcal{Q}_2)$ that is necessary if reachable states of the product are sought. Section 4 proposes an algorithm to perform this efficiently even for large automata.

2.3. LTL Model Checking

Formally, the problem of model checking is deciding $(M, s) \stackrel{?}{\models} \varphi$, where M is a Kripke structure modeling a system, $s \in \mathcal{I}$ is an initial state, and φ is a temporal logic formula describing a desired property of the system [EC80]. The essential idea of LTL model checking is to use Büchi automata to describe both the property and the system model (see Sections 2.2.2 and 2.2.3) and reducing the model checking problem to language emptiness [CGP99].

Given a Kripke structure M and an LTL formula φ using the same atomic propositions AP, let $\mathcal{L}(M)$ and $\mathcal{L}(\varphi)$ denote the languages produced by the Kripke structure and characterized by the formula. The language $\mathcal{L}(M)$ contains every observable behavior of M in terms of AP (provided behaviors), while $\mathcal{L}(\varphi)$ contains valid behaviors. The model checking problem can be rephrased to the following: is the set of provided behaviors fully within the set of valid behaviors? This is called the language inclusion problem and is denoted by $\mathcal{L}(M) \stackrel{?}{\subseteq} \mathcal{L}(\varphi)$.

An equivalent formalization is $\mathcal{L}(M) \cap \overline{\mathcal{L}(\varphi)} \stackrel{?}{=} \emptyset$, where $\overline{\mathcal{L}(\varphi)}$ is the complement of the language of φ. Although both the language inclusion problem and the complementation of Büchi automata are hard, in case of LTL model checking, the complementation can be avoided. The key observation is that the complement of the language of an LTL formula is the language of the negated formula: $\overline{\mathcal{L}(\varphi)} \equiv \mathcal{L}(\neg\varphi)$. This way, the model checking problem is reduced to language intersection and language emptiness, both of which can be efficiently computed on Büchi automata.

This approach is called *automata-theoretic model checking* [Var96]. Given a high-level model and a linear temporal logic specification, the following steps have to be realized (see Figure 1):

1. Compute the Kripke structure M of the high-level model (state space generation).
2. Transform the Kripke structure into a Büchi automaton $\mathcal{A}_M$ (Proposition 1).

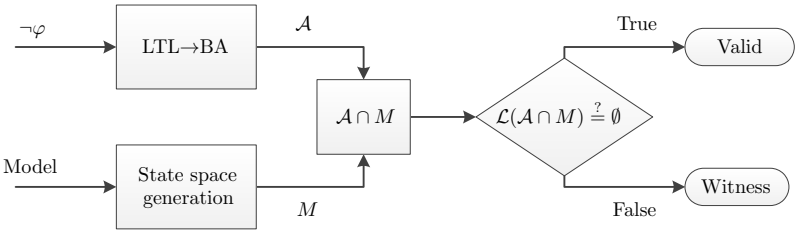


Fig. 1. Steps of ω-regular model checking.

3. Transform the negated LTL formula into a Büchi automaton $\mathcal{A}_{\neg\varphi}$ (Section 2.2.3).
4. Compute the synchronous product $\mathcal{A}_M \cap \mathcal{A}_{\neg\varphi}$.
5. Check language emptiness of the product: $\mathcal{L}(\mathcal{A}_M \cap \mathcal{A}_{\neg\varphi}) \stackrel{?}{=} \emptyset$.

If $\mathcal{L}(\mathcal{A}_M \cap \mathcal{A}_{\neg\varphi}) = \emptyset$ then the model meets the specification. Otherwise words of $\mathcal{L}(\mathcal{A}_M \cap \mathcal{A}_{\neg\varphi})$ are counterexamples, i. e., provided behaviors that violate the specification.

Sometimes, it is possible to design the algorithms such that some steps may overlap or can be executed together. For example, many algorithms compute the product automaton on-the-fly during state space generation, using the high-level model as an implicit representation of the Kripke structure. Language emptiness may sometimes also be checked continuously during the computation of the product. If both optimizations are present in an algorithm, it is said to perform the LTL model checking on the fly (during state space generation) [CVWY91]. For an example, see [GPVW95] that is the base of the SPIN explicit model checker.

2.4. Symbolic Model Checking

Symbolic model checking techniques have been devised to combat the state space explosion problem [McM92, BCM+92]. While traditional *explicit-state model checkers* employ graph algorithms and enumerate states and transitions one-by-one, *symbolic model checking* algorithms use a special encoding to efficiently represent the state space and try to process large numbers of similar states together. The special encoding can be regarded as a compression of the sets and relations, but unlike traditional data compression methods, the encoded data can be manipulated without returning to the explicit representation.

Symbolic model checking was first introduced for hardware model checking, where states of hardware models were encoded in binary variables [CMCH96]. Similarly, states of a Kripke structure can be encoded in at least $k = \lceil log(|\mathcal{S}|) \rceil$ Boolean variables. Sets of states can then be represented by Boolean functions $f_{\mathcal{S}} : \mathbb{B}^k \to \mathbb{B}$, returning true when a state is in the set. The transition relation can also be represented by functions mapping from $2k$ binary variables to true or false, half of them encoding the source state, the other half encoding the target: $f_{\mathcal{R}} : \mathbb{B}^{2k} \to \mathbb{B}$.

The functions are usually represented by Boolean formulae or binary decision diagrams (BDDs). In case of Boolean formulae, the model checking problem is reduced to the Boolean satisfiability problem (SAT), while in case of decision diagrams, efficient algorithms are known to manipulate the sets directly in the encoded form [Bry86]. Based on these operations, model checking can be reduced to fixed point computations on sets of states, such as in the case of saturation presented in Section 3.

2.5. Related Work

Our algorithm uses a hybrid approach that combines symbolic techniques with abstraction and explicit-state model checking. There are a number of related approaches that solve similar problems based on more or less similar techniques.

Explicit-state LTL model checking computes the graph representation of the synchronous product automaton and uses traditional SCC computation algorithms like the one of Tarjan [Tar72] or more recent ones [HPY97]. It provides a natural way to apply model checking on-the-fly, i. e., continuously during the state space traversal [CVWY91, GPVW95] and answer the model checking question without exploring the

full state space in many cases. Explicit-state methods yield the potential of applying various reduction techniques during the traversal such as partial order reduction [Pel98, God96] that is based on cutting redundant orderings of partially ordered actions introduced by the interleaving semantics of the underlying concurrent system models. Partial order reduction is especially efficient for asynchronous, concurrent systems, where state space explosion is a common issue.

Symbolic model checking (discussed in Section 2.4) is used for both CTL (computation tree logic) and LTL model checking. CTL model checking benefits from efficient set manipulation that can be implemented with decision diagram operations (see Section 3.2.1). LTL model checking algorithms were also developed based on decision diagrams and they proved their efficiency [CGH97, STV05]. In these works, the state space and the transition relation of the synchronous product is encoded symbolically, then SCCs satisfying the accepting condition are computed on the synchronous product representation. Symbolic SCC computation based on decision diagrams usually apply greatest fixed point computations on the set of states to compute SCCs [SRB02]. These approaches typically scale well, and they have been improved considerably due to the extensive research in this area. A particularly interesting approach that also employs abstraction is based on compositional refinement introduced in [WBH+06], which uses techniques similar to the one introduced in Section 6.2.

SAT-based methods approach the symbolic model checking problem from a different direction. Traditional SAT-based *bounded model checking* unfolds the state space and the LTL property to a given length (bound), encodes it as a SAT problem and uses solvers to decide the model checking query [BCCZ99]. These approaches are incomplete unless the diameter of the state space is reached. In recent years, new algorithms appeared using induction to provide complete and efficient algorithms for model checking of more complex properties, including LTL [SSS00, McM03, BSHZ11].

A considerable amount of effort was put in combining symbolic and explicit techniques [BZC99, DKPT11a, HIK04, KP08, STV05]. The motivation is usually to introduce one of the main advantages of explicit approaches into symbolic model checking: the ability to look for SCCs on the fly. Solutions typically include abstracting the state space into sets of states such as in the case of multiple state tableaux or symbolic observation graphs. Explicit checks can then be run on the abstraction on the fly to look for potential SCCs.

Our approach builds on these works, as it combines symbolic and explicit techniques too. However, the synchronous product computation is based on new ideas and our approach uses a *series* of fine-grained abstractions instead of a single one to reason about SCCs on the fly. Furthermore, the developed approach employs a novel incremental fixed point computation algorithm to decompose the model checking problem into smaller tasks and incrementally process them.

3. Saturation

The goal of our work is to combine the power of saturation with previous LTL model checking approaches. *Saturation* [CMS03] is a symbolic iteration strategy specifically designed to work with (multivalued) decision diagrams. It was originally used as a state space generation algorithm [CMS06] to answer reachability queries on concurrent systems, but applications in branching-time model checking [ZC09] and SCC computation [ZC11] also proved to be successful. This section is dedicated to introduce the main concepts and algorithms of saturation.

First, the input model formalism of saturation is introduced (Section 3.1). Then the key data structures of saturation, the decision diagrams are defined (Section 3.2). The section continues with the symbolic encoding (Section 3.3) and iteration strategy (Section 3.4) used in saturation. Finally, Section 3.5 presents an extended version of saturation, the constrained saturation algorithm.

3.1. Input Model

Saturation works best if it can exploit the structure of concurrent and asynchronous high-level models. Therefore, it defines the input model on a finer level of granularity, introducing the concept of components and events into traditional transition systems (like Kripke structures).

A component is a part of the model that has its own *local state*. A *global state* is the combination of the local states of every component in the system. In practice, each component i may define a state variable s_i containing its local state, while global states are tuples or vectors of these values denoted by $\mathbf{s} = (s_1, \ldots, s_K)$

(where K is the number of components). A straightforward example for a component in Petri nets can be a single place, with its local state being the number of tokens on it. The global state (the marking of the Petri net) is the combination of all token counts.

An event is a set of transitions somehow related in the high-level model. These transitions typically rely on and change the state of the same components. Again, a transition of a Petri net is a good example of an event. Petri net transitions have well-defined input and output places and they do not rely on or affect any other part of the net. Depending on the current marking, a Petri net transition can represent various transitions of the underlying Kripke structure.

Definition 7 (The input model of saturation) Given a system with K components, the input model of saturation is a 4-tuple $M = \langle \mathcal{S}, \mathcal{I}, \mathcal{E}, \mathcal{N} \rangle$, where:

- $\mathcal{S} \subseteq \mathcal{S}_1 \times \ldots \times \mathcal{S}_K$ is the set of global states with $\mathcal{S}_k$ being the set of possible local states of the kth component;
- $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states;
- $\mathcal{E}$ is the set of events;
- $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$ is the *next-state relation*[2], i. e., the set of allowed state transitions. The set of transitions triggered by a high-level event $\varepsilon \in \mathcal{E}$ is $\mathcal{N}_\varepsilon$, with $\mathcal{N} = \bigcup_{\varepsilon \in \mathcal{E}} \mathcal{N}_\varepsilon$.

The following constructs will often be used throughout the paper: the inverse of the next-state relation $\mathcal{N}^{-1}$, the *next-state function* in the forms $\mathcal{N}(\mathbf{s}) : \mathcal{S} \to 2^{\mathcal{S}}$ and $\mathcal{N}(S) : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$, and the reflexive transitive closure of the next-state relation $\mathcal{N}^*$. The next-state function computes the relational product $S \circ \mathcal{N}$ based on the next-state relation. Considering the next-state relation as a function is often closer to reality, because it is usually given only implicitly by the higher-level model.

3.2. Decision Diagrams

Multivalued decision diagrams (MDDs) [MD98] provide a compact, graph-based representation for functions in the form of $\mathbb{N}^k \to \{0, 1\}$. MDDs are similar to decision trees, but are more compact as the isomorphic subtrees are merged. Also, MDDs can be seen as extensions from binary decision diagrams [Bry86] encoding $\mathbb{B}^k \to \{0, 1\}$ functions. We define MDDs formally as follows.

Definition 8 (Multivalued Decision Diagram) A *multivalued decision diagram* (MDD) encoding the function $f(x_1, x_2, \ldots, x_K)$ (where the domain of each x_i is $D_i = \{0, 1, \ldots, n_i\}$) is a tuple $MDD = \langle V, r, \mathit{level}, \mathit{children}, \mathit{value} \rangle$, where:

- $V = \bigcup_{i=0}^{K} V_i$ is a finite set of *nodes*, where items of V_0 are *terminal* nodes, the rest ($V_N = V \setminus V_0$) are *nonterminal* nodes,
- $\mathit{level} : V \to \{0, 1, \ldots, K\}$ is a function assigning non-negative *level numbers* to each node ($V_i = \{v \in V \mid \mathit{level}(v) = i\}$),
- $r \in V$ is the *root node* of the MDD ($\mathit{level}(r) = K, V_K = \{r\}$),
- $\mathit{children}$: $V_i \times D_i \to V$ is a function defining edges between nodes labeled by items of D_i,
- $\mathit{value} : V_T \to \{0, 1\}$ is a function assigning a *binary value* to each terminal node,

Furthermore, the following well-formedness rules should hold for any MDD. There are exactly two terminal nodes: $V_0 = \{\mathbf{0}, \mathbf{1}\}$, where $\mathbf{0}$ is the terminal zero node ($\mathit{value}(\mathbf{0}) = 0$) and $\mathbf{1}$ is the terminal one node ($\mathit{value}(\mathbf{1}) = 1$). The rest of the nodes are nonterminal nodes ($V_N = V \setminus V_0 = \{v \in V | \mathit{level}(v) > 0\}$). Each nonterminal node $v_k \in V_N$ on level $k = \mathit{level}(v_k)$ has exactly one outgoing edge labeled by each $i \in D_k$ and is determined by $\mathit{children}(v_k, i)$. The target node of the edge outgoing from v_k labeled by $i \in D_k$ is denoted by $v_k[i]$ (i. e., $\mathit{children}(v_k, i) = v_k[i]$). We assume that for each node $v \in V_N$ and $i \in D_{\mathit{level}(v)}$: $\mathit{level}(v) > \mathit{level}(v[i])$, therefore the MDD is *ordered* and the graph determined by V and $\mathit{children}$ is a directed

[2] Early versions of saturation required the Kronecker condition of the next-state relation. This restriction was later removed by encoding the state transitions in decision diagrams rather than matrices. Our solution supports both variants of saturation.

acyclic graph. Also we assume that the MDD is *reduced*, i. e., if $v \in V$ and $w \in V$ are on the same level and all their outgoing edges are the same, then $v = w$.

The semantics (the encoded function f) of such an MDD is the following: $f(a_1, a_2, \dots a_K) = value((((r[a_K])[a_{K-1}]) \cdots)[a_1])$.

3.2.1. Operations and Notations

From a more abstract point of view, decision diagrams can represent sets of n-tuples: encode a function that returns *true* if the given element is in the set and *false* otherwise. Input variables can be obtained by logarithmic (binary) encoding, or by representing the elements naturally by vectors. In this context, it is very useful to define operations on decision diagrams to manipulate the encoded data.

Common operations include the union and the intersection of two sets represented by decision diagrams, complementation, and relational product. The latter is defined for a set A and a relation $R \subseteq A \times B$ as follows: $A \circ R = \{y \mid \exists x \in A, (x, y) \in R\}$. For the description of recursive algorithms realizing union, intersection and complementation with decision diagrams, the reader is referred to [Bry86]. Efficient computation of the relational product is discussed in Section 3. A common property of these algorithms is aggressive caching. Since decision diagrams are obtained by merging isomorphic subtrees of decision trees, many paths in the diagram run into the same subdiagram. Recursive algorithms can cache the results of processing subdiagrams to reuse them when the subdiagram is reached again.

A single node n_k of a decision diagram inherently represents a part of the encoded set. Every path starting in n_k and going to terminal $\mathbf{1}$ represents a partial assignment of some input variables (or vector components). The set of all these *subelements* is denoted by $\mathcal{B}(n_k) \subseteq D_1 \times \dots D_k$ [CLS01]. Paths starting from the root node and ending in n_k also represent a set of subelements denoted by $\mathcal{A}(n_k) \subseteq D_{k+1} \times \dots \times D_K$. The subset of elements that a node n_k encodes in a decision diagram is then $\mathcal{S}(n_k) = \mathcal{B}(n_k) \times \mathcal{A}(n_k)$, corresponding to the set of paths going through n_k and ending in the terminal $\mathbf{1}$. Using the above notations, it is possible to define $\mathcal{B}(n_k)$ recursively:

$$\mathcal{B}(n_k) = \begin{cases} \{i \mid n_k[i] = \mathbf{1}\} & \text{if } k = 1 \\ \bigcup_{i \in D_k} \mathcal{B}(n_k[i]) \times \{i\} & \text{otherwise.} \end{cases}$$

For the sake of convenience, the notation $\mathcal{B}_{[i]}(n_k) = \mathcal{B}(n_k[i]) \times \{i\}$ is introduced to reason about sets of subelements constituting $\mathcal{B}(n_k)$.

3.3. Symbolic Encoding

Saturation works directly on ordered MDDs. It is therefore necessary to define an ordering for the components of the model. Just like in the case of BDDs, the chosen ordering has a crucial impact on the size of the resulting MDD. Furthermore, the efficiency of saturation is also highly dependent on the order of components. There are different heuristics to find a "proper" ordering, such as those introduced in [CLY07, SC06, TMIP04]. The algorithms presented in this paper assume that an ordering is already given.

Without loss of generality, assume that component k is the kth component in the ordering (i. e., components are identified by their indices). By introducing components and events, saturation is able to build on a common property of concurrent systems. *Locality* is the empirical assumption that high-level transitions of a concurrent model usually affect only a small number of components. Locality is exploited both in building a more compact symbolic encoding and in an efficient iteration strategy.

An event $\varepsilon \in \mathcal{E}$ is *independent* from component k if *1)* its firing does not change the state of the component, and *2)* it is enabled independently of the state of the component. If ε depends on component k, then k is called a *supporting* component: $k \in supp(\varepsilon)$. Let $Top(\varepsilon) = max(supp(\varepsilon))$ denote the supporting component of ε with the highest index. Along the value of *Top*, events can be grouped by the highest supporting component: $\mathcal{E}_k = \{\varepsilon \in \mathcal{E} | Top(\varepsilon) = k\}$. For the sake of convenience, $\mathcal{N}_k = \bigcup_{\varepsilon \in \mathcal{E}_k} \mathcal{N}_\varepsilon$ is used to represent the next-state relation of all such events. The self-explanatory notations $\mathcal{E}_{\leq k}$ and $\mathcal{E}_{<k}$ will also be used as well as the corresponding next-state relations $\mathcal{N}_{\leq k}$ and $\mathcal{N}_{<k}$.

Due to the locality property, the next-state relations of events can be decomposed into two parts: *1)* the actual description of local state changes for the supporting variables and *2)* the identity relation for independent variables. Saturation employs a more relaxed decomposition, splitting the relation by the *Top* value:

for an event $\varepsilon \in \mathcal{E}_k$, the next state function is $\mathcal{N}_\varepsilon(s_1, \ldots, s_k, s_{k+1}, \ldots, s_K) \equiv \mathcal{N}_\varepsilon(s_1, \ldots, s_k) \times (s_{k+1}, \ldots, s_K)$, where $\mathcal{N}_\varepsilon(s_1, \ldots, s_k)$ is the projection of the next-state function to components $1 \ldots k$. Thus, it is sufficient to encode the first part, since the event only depends on components lower than the *Top* value.

3.4. Iteration Strategy

The goal of saturation as a state space generation algorithm is to compute the set of reachable states $\mathcal{S}_{rch} = \mathcal{N}^*(\mathcal{I})$ of model M. In other words, the goal is to compute the least fixed point of the next-state function $\mathcal{S}_{rch} = \mathcal{N}(\mathcal{S}_{rch})$ that contains the initial states $\mathcal{I}$. There are many strategies to do this, from simple depth-first or breadth-first explorations to complex iteration strategies such as that of saturation.

The basic idea of the iteration strategy of saturation is also built on the locality property. Instead of computing the fixed point by repeatedly applying the whole next-state function starting from the initial states, the computation is divided into smaller parts. First, only events with the lowest *Top* value are considered, and a *local fixed point* is computed. After that, events with the lowest and the second lowest *Top* values are applied to obtain another fixed point. This process goes on until the global fixed point is reached. Using the notations introduced so far, the difference between classic breadth-first style fixed point computation and saturation's decomposed strategy is shown by the following two (informal) schemes, respectively:

$$\text{Breadth-first style:} \quad \mathcal{S}_{rch} = \mathcal{I} \circ \mathcal{N}^*$$
$$\text{Saturation:} \quad \mathcal{S}_{rch} = ((\ldots((\mathcal{I} \circ \mathcal{N}^*_{\leq 1}) \circ \mathcal{N}^*_{\leq 2}) \circ \ldots) \circ \mathcal{N}^*_{\leq K})$$

Saturation performs these operations on decision diagrams, a fact that allows the definition of the fixed points in terms of decision diagram nodes. A node n_k is *saturated* iff it represents a least fixed point of $\mathcal{N}_{\leq k}$, that is, $\mathcal{B}(n_k) = \mathcal{N}_{\leq k}(\mathcal{B}(n_k))$. An equivalent, recursive definition requires the node to represent the fixed point of $\mathcal{N}_k$ *and* all of its children be saturated.

The iteration strategy follows from the recursive definition: to saturate a node n_k, first saturate all of its children, then apply the next-state function $\mathcal{N}_k$ (i. e., compute the relational product $\mathcal{B}(n_k) \circ \mathcal{N}_k$) iteratively until the fixed point is reached. If new children are created during the latter step, saturate them immediately. The process starts with the root node of the decision diagram representing the set of initial states and recursively calls saturation until it reaches the bottom. Terminal nodes are saturated by definition, so the recursion stops at level 1. By the time the root node is saturated, it represents the set of reachable states.

The power of saturation comes from two sources. First, the algorithm works with smaller next-state relations and decision diagrams, making the local fixed point computations lightweight compared to breadth-first style global fixed point computations. Secondly, like in any decision diagram algorithm, results of the individual computations can be cached to avoid redundant operations.

An extended version of the algorithm's pseudocode will be presented in Section 3.5.

3.5. Constrained Saturation

Constrained saturation is a variant of the saturation iteration strategy introduced in [ZC09], where the state space exploration can be restricted to a given set of states. A motivating example could be a state space exploration algorithm that is allowed to use only states labeled with a certain proposition. To formalize the problem, consider a set of states called the *constraint*: $\mathcal{C} \subseteq \mathcal{S}$. The goal of constrained saturation is to compute the states of $\mathcal{C}$ that are reachable from $\mathcal{I}$ *through states of* $\mathcal{C}$.

The solution is not trivial, since $\mathcal{S} \cap \mathcal{C}$ may contain states that are reachable only through paths not entirely in $\mathcal{C}$. For this reason, the steps of the exploration have to be modified to consider only those target states that are allowed by the constraint: $\mathcal{N}(\mathcal{S}) \cap \mathcal{C}$. Performing an intersection after each step is expensive, so constrained saturation employs a *pre-checking* phase, elements of the next-state relation are sorted out based on whether the target state is in $\mathcal{C}$ or not. Formally, the idea is based on the following observation: $\mathcal{N}(\mathcal{S}) \cap \mathcal{C} = \{\mathbf{s}' \mid \mathbf{s}' \in \mathcal{N}(\mathcal{S}), \mathbf{s}' \in \mathcal{C}\}$.

By using another decision diagram to encode the constraint, the algorithm can locally check the two conditions, i. e., enumerate local successor states and decide whether they are allowed or not. This can be regarded as the evaluation of the binary function that the decision diagram encodes. The constraint is

traversed simultaneously with the decision diagram encoding the set of discovered states, and if it does not contain a path corresponding to the target state, the recursion is stopped on that branch. The pseudocode of the extended algorithm is shown in Algorithms 1 and 2. The extensions compared to the traditional saturation algorithm are marked with an asterisk.

Explanation 1 (Algorithms 1 and 2) *Constrained saturation is implemented in the function* Saturate *that uses the function* RelProd *to compute the relational product of a set of states and a next-state relation. Following the definition of the saturation iteration strategy, both functions are recursive and call each other.*

Saturate *takes a decision diagram node s_k that encodes a set of states and another one c_k encoding the constraint set. The return value is a saturated node. The recursion is terminated by an immediate return if s_k is the "terminal one" node. A call on level k uses the decision diagram representation of the next-state relation $\mathcal{N}_k$ (n_{2k}). A new node (t_k) is created to hold the saturated result, then every non-empty child of s_k is recursively saturated on lines 8–12 (passing the corresponding child of c_k as the new constraint) and copied as a child of t_k. This is the first point where the constraint is checked: local states that are outside of the constraint (i. e., $c_k[i]$ encodes the empty set) are not saturated, but copied as is.*[3]

Lines 13–16 compute the fixed point $\mathcal{B}(n_k) = \mathcal{N}_k(\mathcal{B}(n_k))$. At this point, only the current level k is processed by iterating through the potential local transitions (i, i'). Lower levels are processed by the function RelProd *that takes the lower source states represented by $t_k[i]$, the lower constraint node $c_k[i']$ corresponding to the target state and the lower part of the next-state relation $n_{2k}[i][i']$.* RelProd *ensures that the returned node is always saturated. The result is merged into the existing child of t_k corresponding to the target state i'. Before the recursive call, the constraint is again checked to see whether the target local state is allowed or not.*

Once a fixed point is reached, line 18 ensures that the reduction rules of MDDs hold by replacing node t_k with a previously registered node encoding the same set, if any. Arguments and results are put into a cache to be retrieved when the same subdiagram is reached again on another path.

The role of RelProd *is to recursively compute the relational product, also ensuring that any new node is saturated immediately. The function takes a node s_k encoding the source states, the constraint c_k to be applied on the target states and a third one n_{2k} encoding the next-state relation to be applied on lower levels. The return value is a saturated node including the target states of the given next-state relation.*

Recursion is terminated by an immediate return if s_k and n_{2k} are both the terminal node **1** *(i. e., the algorithm reached the terminal level and the next-state relation could be applied to every level above). If this is not the case, a new node t_k is created to hold the results. Lines 8–10 are the same as lines 14–16 in* Saturate *and have the same goal: recursively compute the relational product. The result is checked to be unique to enforce reduction rules and also saturated. Caching is again employed to avoid redundant computation.*

In an abstract view, the constraint is a series of predicate evaluations on local states – this is the original purpose of decision diagrams (or decision trees). It has a "memory" that can keep track of previous results along the recursive call chain that led the algorithm to the considered decision diagram node. This aspect will be exploited in the algorithm of Section 4.

4. Symbolic Computation of the Synchronous Product

As discussed in Section 2.3, a crucial point of optimization in LTL model checking is the computation of the synchronous product on the fly during state space generation. In this section, a new algorithm is introduced to efficiently perform this step symbolically.

Formally, given a model M defined in Definition 7 and a Büchi automaton $\mathcal{A}$, the task is to directly compute $\mathcal{A}_M \cap \mathcal{A}$ on the fly using saturation, where $\mathcal{A}_M$ is the Büchi automaton accepting the language produced by M as a Kripke structure (described in Proposition 1). Although the result is an automaton, inputs can be omitted from the representation – labels are used only to synchronize the two automata, but are irrelevant in the language emptiness check performed in the next phase of the model checking process.

The algorithm is based on saturation as a state space generation method and reuses the strategy of constrained saturation. The main idea is to decompose the synchronous transitions and to modify constrained saturation to make it compute the possible combinations (see Definition 6). The constraint will serve as a function instead of a set of allowed states, and mechanisms of constrained saturation will be used to evaluate

[3] This can only happen with initial states, which are, by definition, reachable from themselves in zero steps.

Algorithm 1: Saturate

```
   input   : s_k, c_k : node
 1 // s_k: node to be saturated,
 2 // c_k: constraint node
   output  : node
 3 if s_k = 1 then
 4   return 1;
 5 n_2k ← N_k as decision diagram;
 6 Return result from cache if possible;
 7 t_k ← new Node_k;
 8 foreach i ∈ S_k : s_k[i] ≠ 0 do
*9   if c_k[i] ≠ 0 then
10     t_k[i] ← Saturate(s_k[i], c_k[i]);
11   else
12     t_k[i] ← s_k[i];                    // no steps allowed
13 repeat
14   foreach s_k[i] ≠ 0 ∧ n_2k[i][i'] ≠ 0 do
*15     if c_k[i'] ≠ 0 then
16       t_k[i'] ← (t_k[i'] ∪ RelProd(t_k[i], c_k[i'], n_2k[i][i']));
17 until t_k unchanged;
18 t_k ← PutInUniqueTable(t_k);
19 Put inputs and results in cache;
20 return t_k;
```

Algorithm 2: RelProd

```
   input   : s_k, c_k, n_2k : node
 1 // s_k: node to be saturated,
 2 // c_k: constraint node,
 3 // n_2k: next-state node
   output  : node
 4 if s_k = 1 ∧ n_2k = 1 then
 5   return 1;
 6 Return result from cache if possible;
 7 t_k ← new Node_k;
 8 foreach s_k[i] ≠ 0 ∧ n_2k[i][i'] ≠ 0 do
*9   if c_k[i'] ≠ 0 then
10     t_k[i'] ← (t_k[i'] ∪ RelProd(s_k[i], c_k[i'], n_2k[i][i']));
11 t_k ← Saturate(PutInUniqueTable(t_k), c_k);
12 Put inputs and results in cache;
13 return t_k;
```

it on states of the model. This approach is presented in Section 4.1, then Section 4.2 investigates correctness from a formal point of view.

4.1. Encoding the Product Automaton

To use saturation, the synchronous product automaton has to be structured to have components and events according to Definition 7 in Section 3.1. By the definition of such a structure, saturation can be driven to compute the set of reachable states in the product automaton directly while exploring the state space of the system. Formally, the model of the synchronous product $M_\cap$ has to be defined in the form $M_\cap = \langle \mathcal{S}_\cap, \mathcal{I}_\cap, \mathcal{E}_\cap, \mathcal{N}_\cap \rangle$, also requiring the definition of the variable ordering and the set of events.

Saturation is very sensitive to variable ordering, but the relation between the overall performance (runtime and memory usage) and the ordering is very complex and hard to determine. It is usually advised to place related variables[4] close to each other to enhance locality in the structure of decision diagrams as well. This strategy usually reduces the size of the decision diagram encoding of the set of states. In addition, the representation of events also tends to be smaller. Another thing to consider is the *Top* values of events. A great deal of the power of saturation comes from the ability to apply the partitioned next-state relation locally, thus dividing the fixed point computation into smaller parts. This ability is even more enhanced by caching.

Although it is hard to say how much these values affect performance, one corner case is certainly undesirable: Setting every *Top* value to the index of the highest component, K. In this case, saturation would degrade to a somewhat breadth-first style iteration strategy, trying to apply every event on the top level of the decision diagram, effectively flattening the recursive algorithm and degrading cache efficiency in the *Saturate* function.

4.1.1. Encoding the States

Encoding the states of the product is quite natural in the sense that they are pairs $(\mathbf{s}, q) \in \mathcal{S} \times \mathcal{Q}$, which can be represented as vectors, thus it is possible to encode them in a decision diagram. States of the specification

[4] Different definitions of related variables yield different heuristics. For example, variables can be considered related if they are part of the same expression or transitions frequently modify them together.

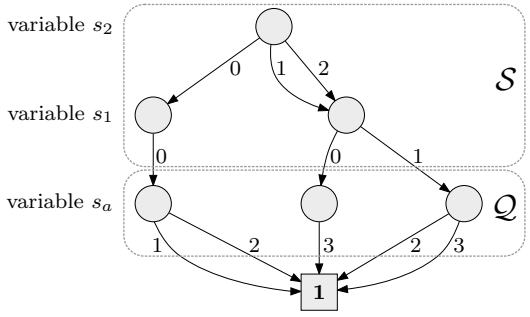


Fig. 2. Example of a decision diagram encoding of a product state space.

automaton can be represented as a vector in any way from using a single variable to binary encoding. For now, assume that the state of the specification automaton is encoded in a single variable $s_a = q$ (i. e., it behaves as a single component in $M_\cap$).

The crucial part is the variable ordering, i. e., the order of the original variables $s_1, \ldots, s_K, s_a$ in the state vector of the product. The following heuristic is based on notations and considerations of Section 3.3 and assumes that the original model M had a variable ordering already optimized for saturation.

Since state transitions of M and $\mathcal{A}$ are synchronized in $M_\cap$, every event of M will trigger a transition in $\mathcal{A}$, i. e., every event of $M_\cap$ will depend on s_a. Formally, $\forall \varepsilon \in \mathcal{E}_\cap, s_a \in \mathit{supp}(\varepsilon)$, and by definition, the value of $\mathit{Top}(\varepsilon)$ might also change. If the *Top* components of events are not to be changed at all (which is only the most straightforward, but not necessarily the best heuristic), putting s_a to the lowest level is an ideal choice. This way, a state of the product is a vector $(\mathbf{s}, q) = (s_a, s_1, \ldots, s_K)$. Figure 2 shows an example of how the encoding decision diagram is structured.

4.1.2. Composing the Transition Relation of the Product Automaton

Decomposing the synchronized transitions means that instead of building a single next-state relation encoding the state changes of both M and $\mathcal{A}$, transitions of the model and the automaton are stored and handled separately. The synchronization itself will be done on-the-fly during the state space traversal by our extended constrained saturation based algorithm.

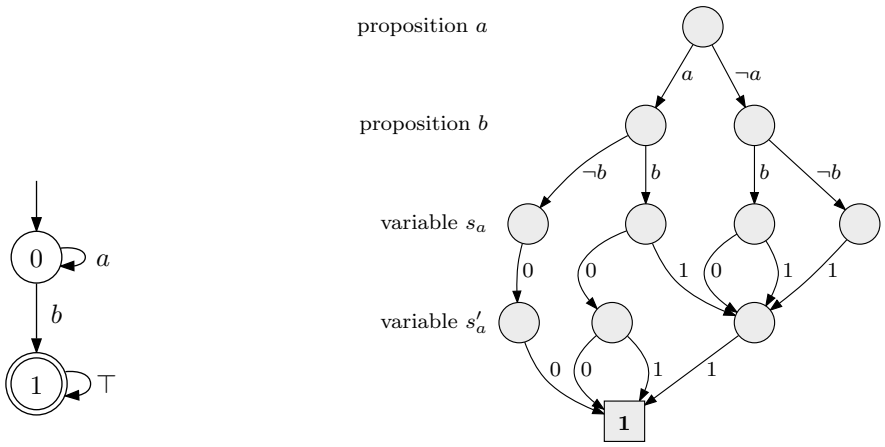
To understand the motivation of the following construct, recall that constrained saturation evaluates a binary function on the states of the system and allows only those that make the function *true*. Also recall that relations can be interpreted as functions, and they can also be encoded in decision diagrams. The idea is to use the "function evaluating" ability of constrained saturation to compute the possible state changes of the automaton based on the labeling of the target states of the system (as defined in Definition 6), according to the transition relation of $\mathcal{A}$.

To do this, the transition relation of $\mathcal{A}$ is reordered to have the signature $\Delta \subseteq 2^{AP} \times \mathcal{Q} \times \mathcal{Q}$. Assuming an ordering of the atomic propositions of AP where $\mathit{ind} : AP \to \{0, \ldots, |AP| - 1\}$ is the indexing of atomic propositions $p \in AP$, a letter $\alpha \in 2^{AP}$ is encoded as a binary vector $\mathbf{p} \in \mathbb{B}^{|AP|}$, where $\mathbf{p}(\mathit{ind}(p)) = \top \Leftrightarrow p = \top$. For an illustration of such an encoding, observe Figure 3(b) that shows the transition relation of a simple Büchi automaton (presented in Figure 3(a)) as a decision diagram.

It is important to note that the ordering of atomic propositions in $\mathbf{p}$ has to be fixed beforehand and also has to match the order of components that are subjects of propositions. The subject of a proposition is a component whose local state is necessary to evaluate the proposition[5] and is denoted by $\mathit{Subject}(p)$. The valuation of an atomic proposition in terms of the local state i of its subject is $p(i) \in \{0, 1\}$.

With the two transition relations defined separately, a synchronous transition of the product will be computed by applying a transition from the selected $\mathcal{N}_k$ on the state of the system, then choosing a "suitable" transition from Δ. The set of "suitable" transitions can be computed from the function representation of automaton transition relation $\Delta : 2^{AP} \to 2^{\mathcal{Q} \times \mathcal{Q}}$ by evaluating the atomic propositions on the target state of

[5] Note that the current version of the algorithm does not support the comparison of variables in different components, so the subject of an atomic proposition is always a single component. In case of bounded variables with the same domain, this restriction can be circumvented by comparing both of them to the same constant value for all possible values in the domain.



(a) Büchi automaton of $[a \mathsf{U} b]$.

(b) Constraint encoding the transition relation of the automaton.

Fig. 3. Minimal Büchi automaton for the LTL formula $[a \mathsf{U} b]$ and its encoding as a constraint.

the system to obtain a letter. Constrained saturation will be used to evaluate the function, with the help of a simple indirection layer evaluating the propositions (see Algorithm 5).

More precisely, as saturation is recursively calling itself and traverses the decision diagram, one of the following can happen. Assume that ℓ is the highest level encoding the automaton ($\ell = 1$ in the pseudocode).

- If the current level belongs to M, i. e., it is above ℓ, the next local transition of $\mathcal{N}$ is used and the constraint evaluates the atomic propositions corresponding to the target local state.
- If the current level is the ℓth, the current constraint node encodes $\Delta(L(\mathbf{s}'))$. For this level and others below it, this relation is used instead of $\mathcal{N}$ to choose the next local transition. The constraint is ignored under this level.
- If the current level is below ℓ (so it belongs to $\mathcal{A}$), the relation $\Delta(L(\mathbf{s}'))$ chosen by α on level ℓ is used to choose the next local transition.

This evaluation step has to be included in both *Saturate* and *RelProd*. The modified version of these functions can be seen in Algorithms 3 and 4.

Explanation 2 (Algorithms 3, 4 and 5) *The modified saturation algorithm for synchronous product computation is implemented in functions* ProdSaturate, ProdRelProd *and* StepConstraint.

StepConstraint *is used to navigate through the "constraint" MDD encoding the transition relation of the automaton. It takes as parameters a node c that represents the current state of the evaluation and two indices i and k, the former encoding the reached local state of the model and the latter identifying the current level (or component).*

The function performs a simple evaluation. If the current level is the lowest (i. e., belongs to the automaton), the function does nothing, but returns c (that may be **0** *or an MDD encoding state transitions of the automaton). Otherwise atomic propositions belonging to level k are evaluated on local state i in the predefined order and the constraint is navigated along the corresponding edges. Note that this navigation may consist of zero or more steps, depending on the number of propositions belonging to level k.*

Functions ProdSaturate *and* ProdRelProd *are very similar to* Saturate *and* RelProd. *There are essentially two differences. First, where constrained saturation navigates the constraint by getting a child of the current node, the modified version uses* StepConstraint *to compute the next constraint node (note that the level of c is now unknown). Secondly, when* ProdRelProd *reaches level* 1, *the node encoding the next-state relation of the model should be* **1** *– this is the point where the constraint is used as the transition relation of the automaton (line 7).*

Formally, the algorithm applies the following set of transitions: $\{((\mathbf{s}, q), (\mathbf{s}', q')) \mid (\mathbf{s}, \mathbf{s}') \in \mathcal{N}, (q, q') \in \Delta\left(L(\mathbf{s}')\right)\}$, i. e., the first part of a transition (originally belonging to the model) takes the model into a state $\mathbf{s}'$, whose labeling is read to choose the second part of the transition (originally belonging to the automaton).

Algorithm 3: ProdSaturate

```
   input  : s_k, c : node
 1 // s_k: node to be saturated,
 2 // c: constraint node
   output : node
 3 if s_k = 1 then
 4   return 1;
 5 n_2k ← N_k as decision diagram;
 6 Return result from cache if possible;
 7 t_k ← new Node_k;
 8 foreach i ∈ S_k : s_k[i] ≠ 0 do
*9   c' ← StepConstraint(c, i, k);
10   if c' ≠ 0 then
11     t_k[i] ← ProdSaturate(s_k[i], c');
12   else
13     t_k[i] ← s_k[i];                  // no steps allowed
14 repeat
15   foreach s_k[i] ≠ 0 ∧ n_2k[i][i'] ≠ 0 do
*16     c' ← StepConstraint(c, i', k);
17     if c' ≠ 0 then
18       t_k[i'] ← (t_k[i'] ∪ ProdRelProd(t_k[i], c', n_2k[i][i']));
19 until t_k unchanged;
20 t_k ← PutInUniqueTable(t_k);
21 Put inputs and results in cache;
22 return t_k;
```

Algorithm 4: ProdRelProd

```
   input  : s_k, c, n_2k : node
 1 // s_k: node to be saturated,
 2 // c: constraint node,
 3 // n_2k: next-state node
   output : node
 4 if s_k = 1 ∧ n_2k = 1 then
 5   return 1;
*6 if k = 1 then
*7   n_2k ← c;                  // transitions of automaton
 8 Return result from cache if possible;
 9 t_k ← new Node_k;
10 foreach s_k[i] ≠ 0 ∧ n_2k[i][i'] ≠ 0 do
*11   c' ← StepConstraint(c, i', k);
12   if c' ≠ 0 then
13     t_k[i'] ← (t_k[i'] ∪ ProdRelProd(s_k[i], c', n_2k[i][i']));
14 t_k ← ProdSaturate(PutInUniqueTable(t_k), c);
15 Put inputs and results in cache;
16 return t_k;
```

Algorithm 5: StepConstraint

```
   input  : c : node i, k: index
 1 // c: constraint node,
 2 // i: index of local state
 3 // k: level of component
   output : node
 4 if k ≤ 1 then
 5   return c ;     // level of automaton, do nothing
 6 foreach p ∈ AP, Subject(p) = k do
 7   c ← c[p(i)] ;                  // evaluate p on i
 8 return c;
```

$\Delta(L(\mathbf{s}'))$ can be seen as a partitioning of the transition relation of the automaton based on the letters that transitions read.

4.2. Correctness and Efficiency

In order to prove the correctness of the algorithm, it is necessary to show that every transition of the product can be simulated by the decomposition-based approach, and no false transitions are introduced by the construct.

Theorem 1 (Correctness of decomposition-based product computation) Given the transition relation $\Delta_\cap$ of the product automaton $\mathcal{A}_M \cap \mathcal{A}$, the next-state relation $\mathcal{N}$ of M and the transition relation Δ of $\mathcal{A}$, every transition of $\Delta_\cap$ can be simulated by the decomposition-based product computation algorithm and vice versa.

Proof. The definition of the transition relation of the product (without the input letters) is $\Delta_\cap = \{((\mathbf{s},q),(\mathbf{s}',q')) \mid (\mathbf{s},\mathbf{s}') \in \mathcal{N}, (q,\alpha,q') \in \Delta, \alpha = L(\mathbf{s}')\}$. The modified constrained saturation applies the transitions $\{((\mathbf{s},q),(\mathbf{s}',q')) \mid (\mathbf{s},\mathbf{s}') \in \mathcal{N}, (q,q') \in \Delta(L(\mathbf{s}'))\}$. By definition, $\Delta(\alpha) = \{(q,q') \mid (q,\alpha,q') \in \Delta\}$, so by substituting $\alpha = L(\mathbf{s}')$ the equivalence follows. □

As the proof suggests, the algorithm directly computes the synchronous transitions on-the-fly without actually storing them. This way, no additional storage is required above the representation of the separate relations, and the computational overhead is also negligible compared to traditional constrained saturation.

5. Symbolic SCC Computation

In this section, the most essential component of the contributed model checking algorithm is presented: algorithms for the detection of strongly connected components (SCCs). As seen in Section 2.3, the LTL model checking problem can be reduced to checking language emptiness, which in turn reduces to the problem of finding fair SCCs in the product state space. This is why devising efficient SCC detection algorithms is especially important here.

The algorithm presented here is correct and complete. However, various heuristics can be used to further improve its efficiency. These extensions will be discussed later in Section 6. The following two sections are an extension of our previous conference paper [MDVB15].

5.1. Main Concepts

There are simple and powerful algorithms for SCC detection in the explicit case, where the state space is represented by a graph that can be traversed freely [Tar72, HPY97]. In a symbolic setting, set operations can be used to compute an SCC-hull as a greatest fixed point in the state space [SRB02]. To introduce the advantage of on-the-fly SCC detection and early termination of explicit techniques into symbolic algorithms, hybrid approaches using abstraction have also spread [HIK04, KP08, DKPT11a]. To further enhance the power of these approaches, both symbolic and explicit methods will be introduced in the following two sections, carefully designed to work together in a symbolic setting.

The symbolic algorithm will be a variant of traditional SCC-hull computation schemes, but optimized to process the state space *incrementally* during the exploration. On the verge of symbolic and explicit, saturation will be enhanced to collect recurring states, states that are visited more than once during the exploration and thus can indicate SCCs. Finally, a cheap abstraction considering decision diagram nodes will also be employed to use explicit algorithms to quickly reason about SCCs in the state space without actually trying to find them. The last two techniques are capable of making the overhead of on-the-fly model checking almost disappear.

Algorithms of this section are orthogonal to the algorithm of Section 4. Every one of them assumes that a model is given in the input format of saturation, no matter if it is a product automaton or anything else. Consequently, the devised methods can be used in settings other than LTL model checking, although in the paper, the focus is on this particular application.

Before presenting the algorithms, a seemingly trivial but fundamental lemma has to be declared. This observation will serve as the basis of the incremental symbolic SCC detection algorithm as well as the explicit and abstraction techniques of the next section.

Lemma 1 (Lemma of saturated state spaces) Given a saturated decision diagram node n_k, any path of $\langle \mathcal{B}(n_k), \mathcal{N}_{\leq k} \rangle$ as a directed graph that is not present in the subspaces $\langle \mathcal{B}_{[i]}(n_k), \mathcal{N}_{<k} \rangle$ (encoded by the children of n_k) contains at least one transition from $\mathcal{N}_k$.

An implication of the lemma is that there is no way to traverse the boundaries of $\mathcal{B}_{[i]}(n_k)$ without using at least one transition from $\mathcal{N}_k$, since the state spaces $\langle \mathcal{B}_{[i]}(n_k), \mathcal{N}_{<k} \rangle$ are *disjoint* (they differ in the local state of component k).

5.2. Incremental Symbolic SCC Computation

In the presented model checking algorithm, on-the-fly SCC detection is intended to be achieved by running SCC computations *frequently* during the state space generation. Certainly, this strategy would mean a massive overhead if repetition of work would not be addressed, so the main requirement towards the design of such an approach is incrementality.

The context of the algorithm is the following. When saturation processes the state space (see Section 4), SCC detection will be run whenever a node n_k becomes saturated. Processing a saturated node has the advantage of handling a set of (sub)states $\mathcal{B}(n_k)$ that is closed with regard to events independent from higher levels ($\mathcal{N}_{\leq k}$). This means that the set will not change anymore during the exploration, i. e., each closed set has to be processed only once.

Even though a set with its related events will be processed only once, the recursive definition of saturation

will cause such sets to appear again as subsets of larger sets encoded by the parent node in the decision diagram (see Section 3.2.1). Due to this, the algorithm has to be able to distinguish parts of the state space already processed (these are $\langle \mathcal{B}_{[i]}(n_k), \mathcal{N}_{<k} \rangle$ encoded by the children nodes) and focus only on new opportunities gained by processing the current node. Since $\mathcal{B}(n_k) = \bigcup_{i \in D_k} \mathcal{B}_{[i]}(n_k)$ and $\mathcal{N}_{\leq k} = \mathcal{N}_{<k} \cup \mathcal{N}_k$ of which $\mathcal{B}_{[i]}(n_k)$ and $\mathcal{N}_{<k}$ are already processed, new elements are in $\mathcal{N}_k$. This is exactly the base idea of the algorithm: in each run, look for only those SCCs that contain at least one transition from $\mathcal{N}_k$.

5.2.1. Elementary Steps of the Fixed Point Computation

The idea can be implemented by discarding transitions from a set of "new" transitions $\mathcal{N}_{new}$ – those that cannot be closed to form a loop. In other words, a transition is discarded if its *source state* cannot be reached from its *target state*. In SCC-hull algorithms [SRB02], sets of states are processed iteratively to eventually get rid of "bad" states (dead-end states, for example) by computing a greatest fixed point of some function on the original set. Now, a set of transitions has to be processed and the function is the following ($\mathcal{N}$ is the set of all transitions, i. e., both "old" and "new" transitions):

$$f(Z) = \{(\mathbf{s}_1, \mathbf{s}_1') \mid (\mathbf{s}_1, \mathbf{s}_1') \in Z, \exists (\mathbf{s}_2, \mathbf{s}_2') \in Z, \mathbf{s}_1 \in \mathcal{N}^*(\mathbf{s}_2')\}$$

The formal goal of the proposed SCC detection algorithm is to compute the greatest fixed point of f as $Z_\Theta = f(Z_\Theta)$ where $Z_\Theta \subseteq \mathcal{N}_{new}$. The following lemma justifies the definition of function f and will prove the correctness and completeness of an incremental step of the SCC detection algorithm.

Lemma 2 (Correctness and completeness) Given a transition system $\langle \mathcal{S}, \mathcal{N} \rangle$ and a set of "new" transitions $\mathcal{N}_{new} \subseteq \mathcal{N}$, the fixed point Z_Θ is *empty* iff $\langle \mathcal{S}, \mathcal{N} \rangle$ does not contain any SCC with transitions from $\mathcal{N}_{new}$.

Proof. The two directions are proven separately.

($\rightarrow$): Indirect proof. Suppose there is a strongly connected component Θ in the transitions system that contains at least one transition from $\mathcal{N}_{new}$, let this be $(\mathbf{s}, \mathbf{s}')$. Also suppose that the fixed point Z_Θ is empty. Consider $f(\{(\mathbf{s}, \mathbf{s}')\})$. By the definition of a strongly connected component, every state of Θ is reachable from every other state of Θ, so $\mathbf{s}$ is also reachable from $\mathbf{s}'$. Reachability in terms of $\mathcal{N}$ is defined by inclusion in the set of states $\mathcal{N}^*(\mathbf{s}')$, so $(\mathbf{s}, \mathbf{s}') \in f(\{(\mathbf{s}, \mathbf{s}')\})$ can be concluded that contradicts the assumption of Z_Θ being empty. Note that because of the transition $(\mathbf{s}, \mathbf{s}')$, Θ is a real (but not necessarily nontrivial) SCC even if $\mathbf{s} = \mathbf{s}'$ because of the self-loop.

($\leftarrow$): The other direction is also proved indirectly. Suppose there is no strongly connected component in the transition system that contains at least one transition from $\mathcal{N}_{new}$. Also suppose that the fixed point is nonempty. Take a transition $\nu_1 = (\mathbf{s}_1, \mathbf{s}_1')$ from Z_Θ. Since it is in the fixed point, its source state $\mathbf{s}_1$ must be reachable from the target state $\mathbf{s}_2'$ of some other transition $\nu_2 = (\mathbf{s}_2, \mathbf{s}_2')$. Now consider this transition and repeat the process. Since Z_Θ is finite, at some point, the transition ν_i will be the same as some transition before: $\nu_i = \nu_j = (\mathbf{s}, \mathbf{s}')$, where $j < i$. At each repetition, the source state of previous transitions were reachable from the target state of the current transition, so $\mathbf{s}_j = \mathbf{s}$ is reachable from $\mathbf{s}_i' = \mathbf{s}'$. Since $\mathbf{s}'$ is obviously reachable from $\mathbf{s}$ through ν, they must be in an SCC, which leads to a contradiction. □

5.2.2. Incremental Steps of the Fixed Point Computation

An incremental step has to compute the fixed point Z_Θ. This is implemented by the function *DetectSCC* that can be seen in Algorithm 6. Checking reachability is performed using saturation, enabling the algorithm to reuse caches in the SCC detection phase as well, just like the decision diagram structures built during the state space exploration.

Compared to the definition of f, the implementation is slightly different. Instead of handling a set of transitions, the implementation reduces the problem to sets of states by considering the source states and target states of every transition in $\mathcal{N}_{new}$. The sets of source and target states are denoted by $\mathcal{S}^-$ and $\mathcal{S}^+$ respectively. Furthermore, a set of states $\mathcal{S}$ (typically the set of states discovered so far) is also input to *DetectSCC* to constrain SCCs – during the state space generation (especially if saturation is used), $\mathcal{N}$ may contain transitions that are not in $\mathcal{S} \times \mathcal{S}$.

The core of *DetectSCC* is the *filtering loop* (lines 5–8), where function f is implemented and performed

iteratively until no more changes occur. In each iteration, the sets $\mathcal{S}^-$ (source states) and $\mathcal{S}^+$ (target states) are filtered:

1. Elements of $\mathcal{S}^-$ that are not reachable from $\mathcal{S}^+$ are removed.
2. Elements of $\mathcal{S}^+$ that are not reachable from $\mathcal{S}^-$ in one step through $\mathcal{N}_{new}$ are removed, ensuring that $\mathcal{S}^-$ and $\mathcal{S}^+$ always contain exactly the source and target states of the remaining transitions.

Lemma 2 still holds if Z_Θ is approximated by $\mathcal{S}^-$ and $\mathcal{S}^+$: transitions are removed from Z when their source states are removed from $\mathcal{S}^-$ in step 1, while target states are removed from $\mathcal{S}^+$ when all of their corresponding transitions got removed from Z to adjust the approximation in step 2. Note that $\mathcal{S}^-$ and $\mathcal{S}^+$ will always be both empty or both nonempty at the fixed point, because a transition has to have a source *and* a target state. However, the iteration can be stopped one step before – if any of them becomes empty, the next step will discard every state from the other one as well.

The number of iterations in the filtering loop has an upper bound of $\mathcal{O}(|\mathcal{N}_{new}|)$, since in every step, at least one transition is discarded from the set. Methods to make the initial set of "new" transitions smaller and thus reduce the number of required steps are discussed in Sections 6.3.1 and 6.3.3.

An incremental step will always assume that there is no SCC in the transition system that *does not contain* a transition from $\mathcal{N}_{new}$ (otherwise the on-the-fly model checking algorithm would have already been terminated). With this, the completeness of the algorithm depends on the strategy of applying the incremental steps.

5.2.3. On-the-fly Search Using the Incremental Steps

The last design question is how and when to call *DetectSCC*. The chosen design is to call *DetectSCC* whenever a node n_k becomes saturated (marked by a circle in Algorithm 7, presenting the final pseudocode of on-the-fly SCC detection). The set of states to constrain the search is $\mathcal{B}(n_k)$, the set of all transitions is $\mathcal{N}_{\leq k}$ and the set of new transitions is $\mathcal{N}_k$ (it is trivial that $\mathcal{N}_k \subseteq \mathcal{N}_{\leq k}$).

In the general case (and in breadth-first style strategies), the whole next-state relation of a model can be partitioned by the traversal strategy, for example, a breadth-first exploration would partition the transitions into "layers" based on their distance from the initial state. If *DetectSCC* is called on each partition as new transitions, Lemma 2 will imply that the algorithm is correct and complete, i. e., it finds an SCC exactly if there exists one.

In saturation, however, the exploration is recursive, so calling *DetectSCC* after a node is saturated does not fall into the above case. Proving that the algorithm is complete can thus be performed inductively.

Theorem 2 (Completeness of incremental SCC detection) Calling *DetectSCC* during state space generation every time a node becomes saturated gives a complete algorithm for deciding if there exists an SCC in a state space, i. e., in at least one call, the fixed point will not be empty iff there exists an SCC in the state space.

Proof. Inductive proof. It is trivial that the empty state spaces encoded by terminal nodes do not contain any SCC. Assume the children of a decision diagram node n_k together with their related transitions $\mathcal{N}_{<k}$ (that is, $\langle \mathcal{B}(n_k[i]), \mathcal{N}_{<k} \rangle$) do not contain SCCs either. Proving that the fixed point will be nonempty iff there exists an SCC in $\langle \mathcal{B}(n_k), \mathcal{N}_{\leq k} \rangle$ would imply that when the root node is saturated and the algorithm stops, the statement of the theorem would hold. The inductive hypothesis directly follows from Lemmas 1 and 2. If there exists an SCC, it must contain a path that is not present in $\langle \mathcal{B}(n_k[i]), \mathcal{N}_{<k} \rangle$ (otherwise $\langle \mathcal{B}(n_k[i]), \mathcal{N}_{<k} \rangle$ would also contain the SCC), so according to Lemma 1 it will contain at least one transition from $\mathcal{N}_k$. This would cause the fixed point to be nonempty (Lemma 2). □

5.3. Extensions to Support Fair SCCs

When looking for fair SCCs, i. e., SCCs containing at least one state from a set of states $\mathcal{F}$, the algorithm can be extended to involve $\mathcal{F}$ as the third set in the filtering loop. Operations performed in the cycle are then the following (the main idea is illustrated in Figure 4):

- Elements of $\mathcal{F}$ that are not reachable from $\mathcal{S}^+$ are removed.

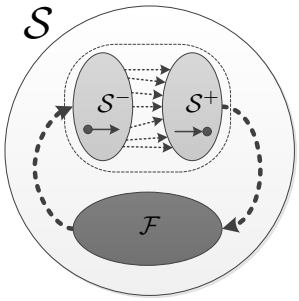


Fig. 4. Illustration of Algorithm 6 extended to look for fair SCCs.

Algorithm 6: DetectSCC

```
   input  : S, N, N_new : set
1  // S: set of states,
2  // N, N_new: set of transitions
   output : bool
3  S⁻ ← N_new^{-1}(S);   S⁺ ← N_new(S⁻);
4  if S⁺ = ∅ then return false;
5  repeat
6  |  S⁻ ← S⁻ ∩ N*(S⁺);
7  |  S⁺ ← S⁺ ∩ N_new(S⁻);
8  until S⁺ and S⁻ unchanged;
9  return S⁻ ≠ ∅ ∧ S⁺ ≠ ∅;
```

- Elements of $\mathcal{S}^-$ that are not reachable from $\mathcal{F}$ are removed.
- Elements of $\mathcal{S}^+$ that are not reachable from $\mathcal{S}^-$ in one step through $\mathcal{N}_{new}$ are removed, ensuring that $\mathcal{S}^-$ and $\mathcal{S}^+$ always contain exactly the source and target states of the remaining transitions.

The first two operations ensure that if a transition of $\mathcal{N}_{new}$ is in an unfair SCC (i. e., does not contain any state from $\mathcal{F}$), it is removed from the fixed point. Note that even accepting trivial SCCs (a single state of $\mathcal{F}$ with a transitions of $\mathcal{N}_{new}$ as a self loop) are found this way, because a state is by definition reachable from itself (i. e., reachability as a relation is reflexive).

When looking for accepting SCCs during LTL model checking, $\mathcal{F}$ is the set of accepting states. Supporting multiple acceptance sets to directly use more complex automata (e. g., generalized Büchi automata) in the model checking algorithm is subject of future work.

6. SCC Computation Made Smart

The SCC computation algorithm presented before is efficient and generally applicable. However, by considering that the goal of any on-the-fly model checking algorithm is to terminate when the first counterexample is found, it is easy to see that at most one SCC detection call is sufficient. In this section, we extend our SCC computation algorithm by adding various methods to *prove the absence* of SCCs and thus prevent unnecessary symbolic fixed-point computations.

When looking for accepting SCCs, checking the absence of accepting states is a usual optimization in similar algorithms, for example in the abstraction refinement approach presented in [WBH+06]. The contribution of this paper goes two steps further. Section 6.1 introduces the use of recurring states and a specialized algorithm to compute them, while Section 6.2 presents new component-wise abstraction techniques tailored to decision diagrams that allow the use of explicit algorithms directly to reason about the presence or absence of SCCs.

The heuristic using recurring states is based on the observation that if no states were seen multiple times during the saturation, then no loop can exist in the state space. The use of abstraction exploits the fact that if an over-approximation of the state space does not contain an SCC, then the state space cannot contain any either. A suitably small abstraction can be used with efficient explicit algorithms that are far cheaper than an actual symbolic detection step.

The new SCC computation workflow is summarized in Figure 5. After presenting the building blocks, Section 6.3 and 6.4 will present the complete SCC algorithm extended with these improvements.

6.1. Recurring States Heuristics

Recurring states are those that have already been discovered before reaching them again during state space exploration. To precisely define them, let a concrete *exploration* ϵ of a fully reachable, connected state space $\langle \mathcal{S}, \mathcal{N} \rangle$ with initial states $\mathcal{I}$ be a sequence of subsets of $\mathcal{S}$, where each element of the sequence contains states discovered in that step: $\epsilon \in (2^{\mathcal{S}})^*$ such that $\epsilon(0) = \mathcal{I}$ and $\epsilon(i+1) \subseteq \mathcal{N}(\mathcal{S}_i)$ for every $0 \leq i < |\epsilon|$, where $\mathcal{S}_i = \bigcup_{0 \leq j \leq i} \epsilon(j)$. An exploration is *full* if $\mathcal{S}_{|\epsilon|} = \mathcal{S}$ and the exploration algorithm considered every enabled transition in $\mathcal{N}$.

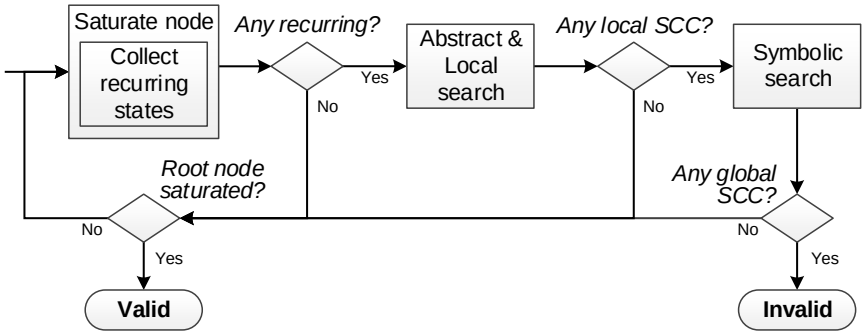


Fig. 5. SCC computation workflow extended with heuristics.

Definition 9 (Recurring states) Given an exploration $\epsilon \in (2^{\mathcal{S}})^*$, the set of recurring states in each step i is $\mathcal{R}_i = \mathcal{S}_i \cap \epsilon(i+1)$, where $\mathcal{S}_i = \bigcup_{0 \leq j \leq i} \epsilon(j)$.

Many explicit algorithms rely on recurring states as indicators of SCCs (for example [Tar72] and [HPY97]). Indeed, they are candidates of being in an SCC, since (apart from the trivially avoidable case of applying the same transitions multiple times) there are only two cases in which they can appear: if the exploration reached them on parallel paths, or a previous state of a path is reached again, i. e., *a loop is found*. Using the following proposition, recurring states can be used to reason about SCCs.

Proposition 2 Given a state space containing at least one SCC, any full exploration will yield at least one state of each SCC as a recurring state.

Using the proposition as an indirect proof, recurring states offer a cheap way to distinguish situations where there is no chance of finding an SCC – situations that often arise during an on-the-fly search.

6.1.1. On-the-fly Collection of Recurring States

Since a basic step in saturation is performed by applying some $\mathcal{N}_\varepsilon$ (i. e., computing the relational product of a set of states and $\mathcal{N}_\varepsilon$), recurring states have to be collected in this granularity as well. In this context, the desired output of the function *RelProd* would be two sets of states: the result of the relational product and the set of recurring states.

Instead of performing the costly intersection at the end of the function, the collection of recurring states can be done on the fly. In general, constrained saturation did exactly the same, as it only allowed steps that stay in a certain set of states. This set is now the set of "old" states, i. e., those that were passed to *RelProd*.

Algorithm 8 shows the implementation of *RelProd* with the collection of recurring states, corresponding lines marked by asterisks. The function basically performs a constrained and an unconstrained saturation simultaneously, gathering the results to two separate sets (decision diagrams). An important note is that contrary to normal constrained saturation, the recursion occurs even if the constraint would not allow the step, because the main functionality is the computation of the relational product.

In each *Saturate* call (Algorithm 7 shows the extended version), all recurring states produced by transitions of a given $\mathcal{N}_k$ are collected separately as the union of sets returned by *RelProd*. Recurring states encountered by applying lower level events during recursive *Saturate* calls are processed locally and will not be considered any further. The arguments of *RelProd* are the original node, the next-state node and the node that represents the relevant part of the old states (with which the results of the relational product will be merged).

It is important to note that this mechanism is useful only if the inputs and results of *RelProdRec* are cached. Otherwise, repeating a previous call would by definition recognize every target state as a recurring state, since they have already been reached by that previous call.

6.2. Using Abstractions to Reason About Emptiness

Hybrid model checking algorithms usually use symbolic encoding to process huge state spaces, while introducing clever abstraction techniques to produce an abstract model on which explicit graph algorithms can be used. In this context, the goal of abstraction is to reduce the size of a system's state space while preserving certain properties, such as the presence or absence of SCCs. The purpose of abstractions is no different in this paper, they are used to reason about SCCs. However, unlike in most approaches in this domain, multiple abstractions are used, ordered in a hierarchy matching the structure of the underlying decision diagram to build an inductive proof about strongly connected components of the state space.

In a symbolic setting, components of the model provide a convenient basis for abstraction. In LTL model checking, it is usual to use the Büchi automaton or its observable language to group states and build an abstraction from these aggregates. The abstraction framework presented in [WBH+06] goes beyond using only one kind of abstraction and explores strategies on a tableau of possible abstractions based on one or more components. More details about using different abstractions in LTL model checking can be found in [BZC99, DKPT11a, HIK04, KP08, STV05].

In addition to selecting the base component, there are multiple ways to define an abstraction in terms of transitions. To illustrate this, two simple abstractions are presented before introducing a new approach of using the structure of a decision diagram to define a more powerful abstraction.

6.2.1. Simple abstractions

Using abstractions to answer binary decisions has two potential goals. One can create an under-approximating abstraction that can say a definite *yes* (these are called *must abstractions*), or an over-approximating one that can say a definite *no* (these are *may abstractions*). To construct an abstraction based on a single component of the system, the definition of an abstraction function is required for both the states and the transitions in the global state space. Abstracting states is straightforward, as the set of local states $\mathcal{S}_k$ of component k can be used directly. Regarding may and must abstractions, different transformations have to be defined for the transitions of the state space.

Definition 10 (May abstraction) May abstraction of a state space $\langle \mathcal{S}, \mathcal{N} \rangle$ based on component k is $\mathcal{A}_k^{\exists} = \langle \mathcal{S}_k, \mathcal{N}_k^{\exists} \rangle$, where $\mathcal{S}_k$ is the set of reachable local states of component k, and $\mathcal{N}_k^{\exists} = \{(s_k, s'_k) \mid \exists \varepsilon \in \mathcal{E}, k \in \mathit{supp}(\varepsilon), \exists((\ldots, s_k, \ldots), (\ldots, s'_k, \ldots)) \in \mathcal{N}_\varepsilon\}$, i. e., the projections of events to component k.

Definition 11 (Must abstraction) Must abstraction of state space $\langle \mathcal{S}, \mathcal{N} \rangle$ based on component k is $\mathcal{A}_k^{\forall} = \langle \mathcal{S}_k, \mathcal{N}_k^{\forall} \rangle$, where $\mathcal{S}_k$ is the set of reachable local states of component k, and $\mathcal{N}_k^{\forall} = \{(s_k, s'_k) \mid \exists \varepsilon \in \mathcal{E}, \mathit{supp}(\varepsilon) = \{k\}, \exists((\ldots, s_k, \ldots), (\ldots, s'_k, \ldots)) \in \mathcal{N}_\varepsilon\}$, i. e., transitions that affect only component k.

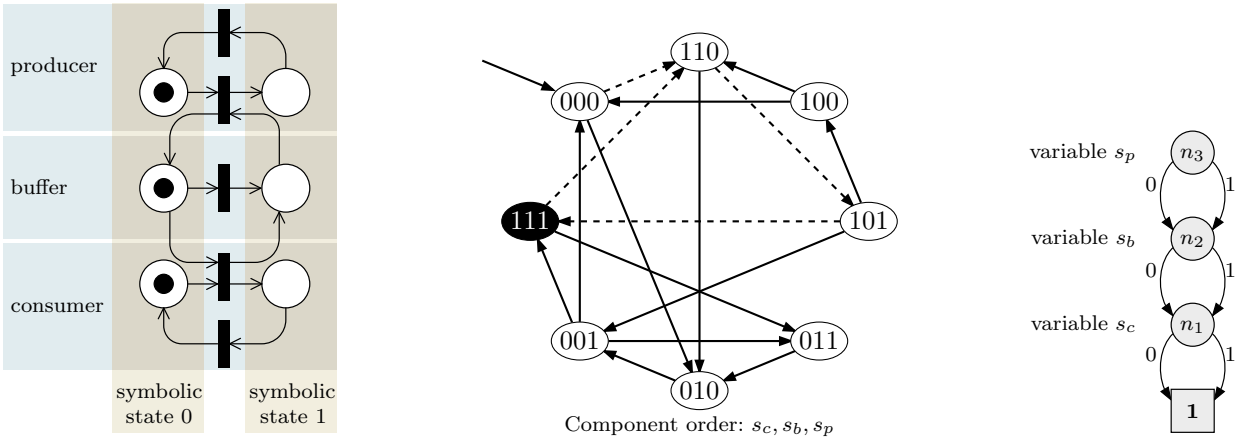
The must abstraction of transitions is defined to keep only those transitions that correspond to events fully within the support of the chosen component. May abstraction preserves every local transition, but omits the synchronization between components (i. e., assumes that if a transition is enabled in component k, it is globally enabled).

Due to these definitions, it is sometimes possible to reason about the presence or absence of global SCCs. If there is an SCC in a single must abstraction, it is the direct representation of one or more SCCs of the global state space. Complementary, if there is no SCC in the may abstraction of *any* component, then the global state space cannot contain any SCC either.

These abstractions usually yield small state graphs that can be represented explicitly. Running linear-time explicit algorithms on them gives a very cheap opportunity to possibly prove or refute the presence of SCCs before symbolic methods are used. Moreover, the definition of may and must abstractions implies $\mathcal{N}_k^{\forall} \subseteq \mathcal{N}_k^{\exists}$, so running the explicit SCC computation on a may abstraction and checking if every transition of a possible SCC is in $\mathcal{N}_k^{\forall}$ effectively considers both cases at the same time.

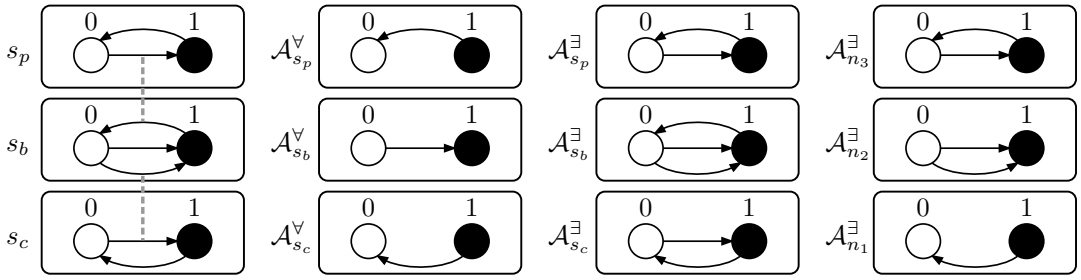
Example 1 *As an example, observe Figure 6. Figure 6(a) illustrates the Petri net model of a producer-consumer system where the buffer has a timeout if the consumer is too slow. The model is divided into three components, with their state variables (s_c for the consumer, s_b for the buffer and s_p for the producer) having two potential values:* 0 *if their token is on the left place and* 1 *if it is on the right. Figures 6(b) and 6(c) show the state space of the model as an explicit state graph and an MDD, with variable ordering s_c, s_b, s_p.*

State transitions of the system are shown in Figure 7(a), with connected arcs representing a single transition affecting multiple components. Assume that every transition of the Petri net defines an event in the



(a) Petri net model. (b) State space. (c) State space MDD.

Fig. 6. Producer-consumer model with non-deterministic buffer.



(a) Original (b) Must (c) May (d) Nodewise

Fig. 7. The effect of the abstractions to the transitions

model. In this case, every state transition belongs to a separate event. Events affecting multiple components can be regarded as synchronization constraints between local transitions. *Abstractions can be acquired by removing synchronizations and local transitions. Figures 7(b) and 7(c) depict the transitions transformed by must and may abstractions, respectively, for every component. If the goal is to find an SCC containing the state where only the places on the right of the Petri net are marked (depicted as a black state in Figure 6(b)), none of the simple abstractions of any component can provide information about SCCs.*

6.2.2. Node-wise abstraction

Among the main algorithmic contributions of this paper, the last one is a specialized abstraction that fits saturation and the presented incremental symbolic SCC detection algorithm, as well as it complements Proposition 2 (about recurring states) as a cheap way to prove the absence of SCCs. The goal of the following construct is to match the order in which events are processed during saturation, as well as the structure of the underlying decision diagram.

Definition 12 (Node-wise abstraction) Node-wise abstraction of state space $\langle \mathcal{S}, \mathcal{N} \rangle$ with regard to node n_k is $\mathcal{A}^{\exists}_{n_k} = \langle \mathcal{S}_{n_k}, \mathcal{N}^{\exists}_{n_k} \rangle$, where $\mathcal{S}_{n_k} = \{i \mid \mathcal{B}(n_k[i]) \neq \emptyset\}$, i. e., reachable local states[6] encoded by the arcs of n_k, and $\mathcal{N}^{\exists}_{n_k} = \{(s_k, s'_k) \mid s_k, s'_k \in \mathcal{S}_{n_k}, \exists((\ldots, s_k, \ldots), (\ldots, s'_k, \ldots)) \in \mathcal{N}_k\}$, i. e., the projections of events $\mathcal{E}_k$ to component k.

[6] In a fully reduced MDD, $\mathcal{B}(n_k[i]) \neq \emptyset \Leftrightarrow n_k[i] \neq \mathbf{0}$.

Node-wise abstraction can be regarded as a may abstraction of the states space $\langle \mathcal{B}(n_k), \mathcal{N}_{\leq k} \rangle$ based on component k. Just like may abstractions, a series of node-wise abstractions can also be used to reason about global SCCs. Moreover, this can be done inductively during the state space generation. The following theorem gives the basis for this inductive method of using node-wise abstractions to prove the absence of SCCs.

Theorem 3 (Node-wise abstraction and SCCs) Given a node-wise abstraction $\mathcal{A}^{\exists}_{n_k}$ with regard to a saturated node n_k, the state space $\langle \mathcal{B}(n_k), \mathcal{N}_{\leq k} \rangle$ does not contain any SCC if the following assumptions hold: *1)* neither the abstract state graph $\mathcal{A}^{\exists}_{n_k}$ *2)* nor the state spaces $\langle \mathcal{B}_{[i]}(n_k), \mathcal{N}_{<k} \rangle$ belonging to the children of n_k contain an SCC.

Proof. Indirect proof. Suppose that $\langle \mathcal{B}(n_k), \mathcal{N}_{\leq k} \rangle$ contains an SCC with a state in $\mathcal{B}_{[i]}(n_k)$ (i. e., component k is in local state i). There are three possible cases to realize this:

1. The SCC is fully within $\mathcal{B}_{[i]}(n_k)$ and contains transitions only from $\mathcal{N}_{<k}$;
2. The SCC is fully within $\mathcal{B}_{[i]}(n_k)$ and contains transitions from $\mathcal{N}_k$, but they do not change the local state of component k;
3. The SCC contains a state from at least one other $\mathcal{B}_{[j]}(n_k)$ $(i \neq j)$ as well.

Case 1 contradicts Assumption 2, while case 2 is also a contradiction with Assumption 1, since not changing the local state of component k would mean a self loop in $\mathcal{A}^{\exists}_{n_k}$ constituting a trivial SCC. Proving that case 3 also yields a contradiction is based on Lemma 1. To reach the state in $\mathcal{B}_{[j]}(n_k)$ from the state in $\mathcal{B}_{[i]}(n_k)$, a path has to use at least one transition from $\mathcal{N}_k$. These transitions will also form a path in $\mathcal{A}^{\exists}_{n_k}$ from i to j. To be in an SCC, there has to be a path from the state in $\mathcal{B}_{[i]}(n_k)$ to the state in $\mathcal{B}_{[j]}(n_k)$ also using at least one transition from $\mathcal{N}_k$, which in turn also forms a path in $\mathcal{A}^{\exists}_{n_k}$ from j to i. Since i and j are reachable from each other, they are in an SCC of $\mathcal{A}^{\exists}_{n_k}$, which is again a contradiction with Assumption 1. □

The main idea of the proof is that node-wise abstraction represents the effects of the events $\mathcal{E}_k$ exactly on the level of their *Top* value. At the time a node n_k becomes saturated, the only transitions that can change the local state of component k are in $\mathcal{N}_k$. Node-wise abstractions contain the images of exactly these transitions, thus they describe the possible transitions between sets of substates encoded by the children of n_k. This is why they can be used to identify so-called *one-way walls* in the state space that separate the possible spaces for SCCs.

Note that the theorem did not specify how to ensure Assumption 2 (SCC-free state spaces of children nodes). This means that the algorithm is not restricted to using node-wise abstraction to prove SCC-freeness for the children – even if the corresponding node-wise abstraction did contain an SCC (which only implies the *possible* presence of a global SCC), the symbolic fixed point computation algorithm of Section 5.2 can check if the abstract SCC candidate is realizable in the global state space or not. This way, the series of saturated nodes can inductively prove the absence of SCCs by the end of the state space generation.

Example 2 *Observing Figure 7 again, node-wise abstractions of the state space with regard to the three decision diagram nodes can be seen in Figure 7(d). As Theorem 3 suggests, it is unnecessary to start symbolic SCC detection until the top level node n_3 (corresponding to the consumer component) gets saturated, since it is the only node whose node-wise abstraction contains an SCC.*

By the time a node is saturated, its node-wise abstraction will not change anymore. This way, a single abstraction has to be built and analyzed only once. The computation of node-wise abstractions is very simple and cheap. It can be done on demand by projecting the next-state relation of corresponding events to the *Top* component, or on-the-fly during saturation by adding vertices and arcs each time a new local state is discovered or a new transition of the corresponding events is fired, respectively (marked by a diamond in Algorithm 7). A simple must abstraction can also be examined as part of computing SCCs of the node-wise abstraction by checking if every local transition used in the SCC belongs to events having only the current component as a supporting one. If this is the case, that SCC is inherently realizable and can be returned as a counterexample immediately.

6.3. On-the-fly Incremental Hybrid Model Checking of LTL Properties

In this section, the building blocks presented so far are assembled into an on-the-fly, hybrid and incremental LTL model checking algorithm. Section 6.3.1 will show a way to reuse recurring states and the modified relational product operator in the symbolic fixed point computation algorithm. Section 6.3.2 shows how to consider recurring and accepting states in the explicit search, while Section 6.3.3 will integrate the symbolic and explicit searches. Section 6.4 will present the full hybrid SCC detection algorithm.

6.3.1. Recurring States in the Fixed Point Computation

In addition to indicating the absence of SCCs, recurring states can also help in finding them. Since each *Saturate* builds its own set of recurring states, the set of all gathered recurring states on level k will all be target states of transitions in $\mathcal{N}_k$ (only these transitions are fired on level k). This way, the "interesting" subset of $\mathcal{S}^+$ is available at the end of a *Saturate* call and *DetectSCC* can be initialized with the recurring states instead of $\mathcal{B}(n_k)$.

Corollary 1 (Accelerating the fixed point computation) During an SCC detection phase after the saturation of node n_k, restricting the set of "new" transitions $\mathcal{N}_k$ to those that end in recurring states (that is, $\{(\mathbf{s},\mathbf{s}') \mid (\mathbf{s},\mathbf{s}') \in \mathcal{N}_k, \mathbf{s}' \in \mathcal{R}\}$) will still find all SCCs that would be found otherwise.

The corollary follows from the fixed point computation strategy of saturation and its caching mechanisms. It can be used to accelerate the fixed point computation by using a smaller input set, since a smaller set of "new" transitions makes *DetectSCC* finish in fewer iterations.

Another application of *RelProd* is the computation of the filtered sets in the filtering loop of *DetectSCC*. Since *RelProd* can be used to compute the intersection of the relational product and any other set of states on the fly, it is suitable to substitute the intersection used in the computation of reachable states.

6.3.2. Constraining the Explicit Search of Abstractions

Recurring states together with accepting states (if any) are useful in the explicit search on node-wise abstractions as well. According to Proposition 2, every SCC has to contain at least one recurring state. If fair SCCs are sought, then at least one state of an acceptance set $\mathcal{F}$ also has to be included in the SCC.

In a node-wise abstraction $\mathcal{A}^{\exists}_{n_k}$, every node i represents the set of states $\mathcal{B}_{[i]}(n_k)$. If the SCC candidate is realizable, it will contain some states from these sets. A necessary property of abstract SCC candidates is therefore described by the following corollary that is a direct consequence of the above considerations.

Corollary 2 (Necessary condition for abstract SCC realizability) An abstract candidate SCC $\Theta_{\mathcal{A}} = \langle \mathcal{S}_\Theta, \mathcal{N}_\Theta \rangle$ of a node-wise abstraction $\mathcal{A}^{\exists}_{n_k}$ is not realizable or not fair if

- $\left(\bigcup_{i \in \mathcal{S}_\Theta} \mathcal{B}_{[i]}(n_k)\right) \cap \mathcal{R} = \emptyset$, or
- $\left(\bigcup_{i \in \mathcal{S}_\Theta} \mathcal{B}_{[i]}(n_k)\right) \cap \mathcal{F} = \emptyset$, respectively.

Since every $\mathcal{B}_{[i]}(n_k)$ as well as $\mathcal{R}$, and even the set of accepting states (formally $\{(\mathbf{s}, q) \mid q \in \mathcal{F}\}$) is known by the time of the explicit search, candidate SCCs can be evaluated and sorted out if the necessary conditions do not hold. If no potentially realizable candidate SCCs remain, the state space does not contain any global SCCs either.

6.3.3. SCC Candidates in the Fixed Point Computation

Node-wise abstraction and the incremental symbolic fixed point computation algorithm are strongly related. Node-wise abstraction contains exactly those transitions that are considered "new" in the fixed point algorithm, so the latter can be regarded as a method to check whether a candidate SCC is realizable or not.

According to Definition 12 (the definition of node-wise abstraction) and Theorem 3, arcs of candidate SCCs represent the transitions that *may* be part of an SCC – if an arc is not part of any candidate SCC, its corresponding transitions will not be part of a global SCC either.

Corollary 3 (Realizing abstract SCCs) Given a node-wise abstraction $\mathcal{A}^{\exists}_{n_k}$ and an abstract SCC candidate $\Theta = \langle \mathcal{S}_\Theta, \mathcal{N}_\Theta \rangle$, the only transitions of $\mathcal{N}_k$ that can be part of the global SCC are those corresponding

to abstract arcs $\mathcal{N}_\Theta$, formally $\{(\mathbf{s},\mathbf{s}') \mid (\mathbf{s},\mathbf{s}') \subseteq \mathcal{N}_k, (s_k, s'_k) \in \mathcal{N}_\Theta\}$, where s_k and s'_k are the local states of component k in the global states $\mathbf{s}$ and $\mathbf{s}'$.

The corollary can be exploited in the symbolic fixed point algorithm by considering only those transitions as "new" that are part of a candidate SCC instead of the full relation $\mathcal{N}_k$.

6.4. Assembling the Pieces – The Full SCC Detection Algorithm

After introducing the different aspects and components of the incremental hybrid SCC detection algorithm presented in this section, the last section summarizes the algorithm as a whole. The input model is assumed to be in any form saturation can process. SCC detection relies only on a base algorithm employing the iteration strategy of saturation, including, but not limited to traditional saturation, constrained saturation or the product computation algorithm of Section 4.
Whenever a node n_k becomes saturated, the following steps have to be executed:

1. The set of encoded states $\mathcal{B}_{[i]}(n_k)$ and transitions $\mathcal{N}_k$ are checked against emptiness – if any of them is empty, SCC detection reports "no SCC".
2. The set of collected recurring states $\mathcal{R}$ and – if applicable – accepting states $\mathcal{F}$ are checked against emptiness – if no recurring states were found, SCC detection reports "no SCC" (see Proposition 2).
3. An explicit SCC computation algorithm is run on the current node-wise abstraction $\mathcal{A}^{\exists}_{n_k}$ to obtain an SCC candidate – if no candidate is found, SCC detection reports "no SCC" (see Theorem 3).
4. Candidate SCCs are checked according to Corollary 2 – if none of them is realizable, SCC detection reports "no SCC".
5. A fixed point computation is started with $\mathcal{S}^+ = \mathcal{R}$ and transitions corresponding to arcs of the candidate SCCs as "new" transitions – the result of the computation is returned as the result of this iteration (see Lemma 2, Theorem 2 and Corollaries 1 and 3).

As summarized in Figure 5, the algorithm terminates as soon as *DetectSCC* finds a nonempty fixed point, or if the root node is saturated. If a nonempty fixed point is found, its contents can be used to aid counterexample generation (e. g., saturation-based SCC computation algorithms of [ZC11]). Otherwise, there is no counterexample and the model M is valid in terms of the specification φ.

Algorithms 7 and 8 show the pseudocodes of the modified *Saturate* and *RelProd* functions.

Explanation 3 (Algorithms 7 and 8) *SCC detection with saturation is implemented in functions* SaturateSCC *and* RelProdSCC. *They are very similar to* Saturate *and* RelProd, *but without a constraint. Differences are marked by an asterisk in case of lines related to recurring states, a diamond marks the on-the-fly construction of the node-wise abstraction, and a circle shows the line where symbolic SCC detection is called.*

RelProdSCC *has two additional parameters:* o_k *is a node encoding the "old states" that the return value will be merged with (in order to compute their intersection) and* r_k *is an in-out parameter used to return this intersection (the recurring states). Computation of this intersection is integrated with the function* RelProdSCC. *When the recursion ends with reaching* $\mathbf{1}$, o_k *is checked. If it is* $\mathbf{1}$, *then the found new state is in fact already in the set of states that will be merged with the result, so* r_k *is also set to* $\mathbf{1}$ *to include the current state in the set of recurring states.*

In order to collect recurring states, a node r_k *is created in every* SaturateSCC *call. During the computation of the relational product, another temporary node is passed to every call of* RelProdSCC *as an in-out parameter to collect the recurring states of the lower levels. When the function returns, this node is merged into the corresponding child of* r_k. RelProdSCC *also collects recurring states this way. Once* t_k *is saturated, the algorithm checks* r_k *to see if there is any recurring state collected.*

The abstraction is built on the fly every time a local transition processed in SaturateSCC *is found to be globally fireable, i. e., the* RelProdSCC *returns a nonempty node. If recurring states are found, transitions that yielded an SCC in* $\mathcal{A}^{\exists}_{t_k}$ *are computed*[7] *and stored in* $\mathcal{N}_\Theta$.

Finally, DetectSCC *is called with the set of recurring states* $\mathcal{B}(r_k)$, $\mathcal{N}_{\leq k}$ *as the set of all processed*

[7] This step can also be done on the fly inside *DetectSCC*.

Algorithm 7: SaturateSCC

```
   input   : s_k : node
 1 // s_k: node to be saturated,
   output  : node
 2 if s_k = 1 then
 3   return 1;
 4 n_2k ← N_k as decision diagram;
 5 Return result from cache if possible;
 6 t_k ← new Node_k;
 7 foreach i ∈ S_k : s_k[i] ≠ 0 do
 8   t_k[i] ← SaturateSCC(s_k[i]);
*9 r_k ← new Node_k;
10 repeat
11   foreach s_k[i] ≠ 0 ∧ n_2k[i][i'] ≠ 0 do
*12     r'_{k-1} ← new Node_{k-1};
13     u_{k-1} ← (RelProdSCC(t_k[i], n_2k[i][i'], t_k[i'], r'_{k-1}));
◇14     if u_{k-1} ≠ 0 then add arc (i, i') to A^∃_{t_k};
15     t_k[i'] ← (t_k[i'] ∪ u_{k-1});
*16     r_k[i'] ← (r_k[i'] ∪ r'_{k-1});
17 until t_k unchanged;
*18 r_k ← PutInUniqueTable(r_k);
19 t_k ← PutInUniqueTable(t_k);
20 Put inputs and results in cache;
*21 if r_k = 0 then return t_k;
◇22 N_Θ ← transitions corresponding to SCCs of A^∃_{t_k};
○23 if DetectSCC(B(r_k), N_≤k, N_Θ) then
24   terminate with counterexample;
25 return t_k;
```

Algorithm 8: RelProdSCC

```
   input   : s_k, n_2k, o_k : node
 1 // s_k: node to be saturated,
 2 // n_2k: next-state node,
 3 // o_k: old node,
   in-out  : r_k : node
 4 // r_k: recurring states
   output  : node
 5 if s_k = 1 ∧ n_2k = 1 then
*6   if o_k = 1 then r_k ← 1;
 7   return 1;
 8 Return result from cache if possible;
 9 t_k ← new Node_k;
10 foreach s_k[i] ≠ 0 ∧ n_2k[i][i'] ≠ 0 do
*11   r'_{k-1} ← new Node_{k-1};
12   t_k[i'] ← (t_k[i'] ∪ RelProdSCC(t_k[i], n_2k[i][i'], t_k[i'], r'_{k-1}));
*13   r_k[i'] ← (r_k[i'] ∪ r'_{k-1});
*14 r_k ← PutInUniqueTable(r_k);
15 t_k ← SaturateSCC(PutInUniqueTable(t_k));
16 Put inputs and results in cache;
17 return t_k;
```

transitions and $\mathcal{N}_\Theta$ as the set of new transitions.[8] *If it returns true (i. e., an SCC is detected), the whole algorithm is terminated with a counterexample.*

7. Evaluation

To demonstrate the efficiency of the presented new model checking algorithm (referred to as *Hyb-MC*), models of the Model Checking Contest[9] have been used to compare it to three competitive tools. NuSMV 2 [CCG+02] is a BDD-based model checker implementing traditional SCC-hull algorithms and is well-established in the industrial and academical community. Its successor, nuXmv [CCD+14] –among other methods– implements the IC3 algorithm for LTL model checking. ITS-LTL [DKPT11b] is a powerful tool based on saturation that implements various optimizations both for the symbolic encoding and on-the-fly SCC detection.

7.1. Implementation

The algorithms presented in Section 4–6 were implemented in the PetriDotNet[10] framework. PetriDotNet is a modeling and model checking tool written in C# supporting colored and ordinary Petri nets. It is developed by students of Fault Tolerant Systems Research Group of Budapest University of Technology and Economics, including two authors of this paper. Currently (in 2015), it supports basic analysis of Petri nets, and a handful of saturation-based model checking techniques including state space generation and reachability analysis, CTL model checking, bounded CTL model checking and on-the-fly LTL model

[8] Checking the emptiness of $\mathcal{N}_\Theta$ is not necessary, because *DetectSCC* will terminate almost immediately if $\mathcal{N}_{new}$ is empty.
[9] http://mcc.lip6.fr/
[10] http://petridotnet.inf.mit.bme.hu/en/

checking (presented in this paper). Here we only use it as an interface to models, and we reuse the basic saturation algorithms and related data structures from the previous CTL model checking algorithms. The implementation of our new method also uses tools of the SPOT toolset [DP04] to parse LTL formulae and transform them to Büchi automata.

7.2. Benchmark Cases

The Model Checking Contest offers Petri net models of various artificial and real-world problems. The models are given in PNML format [HKK+09], usually both as colored Petri nets and unfolded Petri nets (described in Section 2.1.2) as well. Due to the supported input formats of the selected tools, only ordinary Petri nets could be used.

Excluding the "surprise models" of the contest of 2014 (which were released after conducting the measurements), a total of 27 scalable models were used in the benchmark, with instances of different size, resulting in 157 model instances. The majority of the models expose concurrent and asynchronous behavior. Except the "planning" model, state spaces of the nets are finite. Even the infinite model was kept in order to demonstrate that on-the-fly LTL model checking can sometime bear with infinite models: if the product is finite or a counterexample is found in a finite subspace.

As for the specifications, a tool of the SPOT toolset was used to generate real (i. e., no pure Boolean) LTL formulae with predefined atomic propositions. Atomic propositions were generated based on the models: for every place of the smallest instance (those that appear in instances of any size) two propositions requiring zero and nonzero token counts were defined. SPOT generated 50 properties for every model class – instances of every size were checked against these properties. All in all, the 50 properties for each of the 157 model instances gave a total of 7 850 benchmark cases.

7.2.1. Generated LTL Formulae

There are many categorizations of LTL formulae based on syntactic or semantic considerations [MP92]. The generated LTL properties were also categorized by a tool of SPOT. The following categories were used in the measurements.[11]

- *Safety properties* (1 466 formulae): Specifies that something "bad" never happens. In general, counterexamples of these properties consist of a finite prefix that cannot be "fixed" with any suffix, i. e., the violation occurs in the prefix itself. In LTL, they are usually in the form $\mathsf{G}\ \varphi$.
- *Guarantee properties* (2 112 formulae): Specifies that something "good" is guaranteed to happen. Counterexamples for guarantees are "lasso" shaped, since they have to describe an infinite behavior that fails to expose the desired property. In LTL, they are usually in the form $\mathsf{F}\ \varphi$.
- *Obligation properties* (4 975 formulae): Combination of safety and guarantee properties. In LTL, they are usually in the form $\mathsf{G}\ \varphi_1 \vee \mathsf{F}\ \varphi_2$.
- *Pure eventuality formulae* (1 620 formulae): If φ is a pure eventuality formula and the path ρ models φ, then $\zeta\rho$ also models φ, where $\zeta \in \mathcal{S}^*$ is any finite prefix. In other words, pure eventualities describe *left-append closed* languages.
- *Pure universality formulae* (1 260 formulae): If φ is a pure universality formula and the path $\zeta\rho$ models φ, then ρ also models φ, where $\zeta \in \mathcal{S}^*$ is any finite prefix. In other words, pure universalities describe *suffix closed* languages.

There are some more complex and interesting categories that cannot be automatically recognized by SPOT. For this reason, the category "not obligation" (2 875 formulae) is used for such properties, including the following categories.

- *Progress properties*: Specifies that something "good" will keep happening again and again. In LTL, they are usually in the form $\mathsf{G}\ \mathsf{F}\ \varphi$.
- *Response properties*: Specifies that a response will eventually be given to a certain event whenever it occurs. In LTL, they are usually in the form $\mathsf{G}\ \varphi_1 \Rightarrow \mathsf{F}\ \varphi_2$.

[11] The categories are not always exclusive. For example, obligation properties include safety and guarantee properties.

- *Stability properties*: Specifies that a certain property will stabilize eventually. In LTL, they are usually in the form $\mathsf{F}\,\mathsf{G}\,\varphi$.

  As it turned out, the presented algorithm is not sensitive to the category of the checked LTL property.

7.2.2. Tools and Inputs

As mentioned, the tools selected for comparison are NuSMV, nuXmv and ITS-LTL. Unfortunately, PNML is not supported in any of the tools selected for comparison. Both NuSMV and nuXmv consume models in their native SMV format, while ITS supports many types of models and formats other than PNML. In case of NuSMV and nuXmv, PNML models were translated to SMV by our model exporter in PetriDotNet, similarly to the approach discussed in [SBJ14]. Since ITS accepts models in CAMI format (the native format of the tool CPN-AMI) and there exists a widely-used tool to convert between PNML and CAMI, this tool was used to generate the input files for ITS.[12]

In terms of the properties, both Hyb-MC and ITS-LTL uses SPOT to parse the formula and transform it into a Büchi automaton. A small tool was implemented to transform these formulae into the format of NuSMV and nuXmv.

7.2.3. Benchmark settings

Measurements were done on identical server machines with Intel Xeon processors (4 cores, 2.2GHz) and 8 GB of RAM. Hyb-MC and ITS-LTL were run on Windows 7 x64, while NuSMV and nuXmv were run on CentOS 6.5 x64.

The timeout of each measurement was set to 600 seconds. Runtimes were measured internally by every tool. In case of Hyb-MC, the runtime includes the processing time of SPOT (transformation to automaton) and the total runtime of the algorithm including its initialization, but not the loading time of PetriDotNet. ITS-LTL also measured its runtime internally, also including the runtime of SPOT and the algorithm. NuSMV and nuXmv can measure time by placing a special command in the input script. In case of these tools, only the actual runtime of model checking was measured, omitting the time of loading the model and constructing the binary model from the SMV input, making their measured runtimes slightly smaller than the actual runtime. In total, approximately 40 days of processing time was spent on the evaluation.

The decision diagram based tools (Hyb-MC, NuSMV and ITS-LTL) used the same variable ordering produced by heuristics of the ITS toolset. Out of the numerous Python scripts bundled with ITS-LTL, "Script 11" was used to produce a flat ordering with each place encoded in a separate variable – this produced orderings that every tool could parse and use. This way, the hierarchical features of ITS might not have been fully exploited, but saturation-based algorithms in PetriDotNet can also be faster if a variable can encode multiple places.

7.3. Results

The following sections present some aspects of the results of the evaluation. Overall, 5 megabytes of measurement log was collected and analyzed. Out of the successfully checked cases, properties were fulfilled 2 811 times, while 3 565 cases gave counterexamples. In 1 474 cases, all the tools exceeded the predefined time limit. The whole benchmark and the analyzed results can be downloaded from the website[13] of PetriDotNet dedicated to [MDVB15].

7.3.1. Comparison of Runtime Results

The main emphasis of preliminary analysis was on the runtime results of each tool. The runtime of Hyb-MC was compared to the corresponding runtime of the other tools. In addition, simple state space generation

[12] The conversion between different input formats might have an impact on the performance of the tools. While ITS can handle Petri nets directly (only the file format has to be changed), the models have to be transformed into SMV format for the NuSMV and nuXmv tools. Direct, manual modeling could have yielded a more efficient model, but due to the number of benchmark models, automatic conversion had to be used.

[13] `http://inf.mit.bme.hu/en/tacas15`.

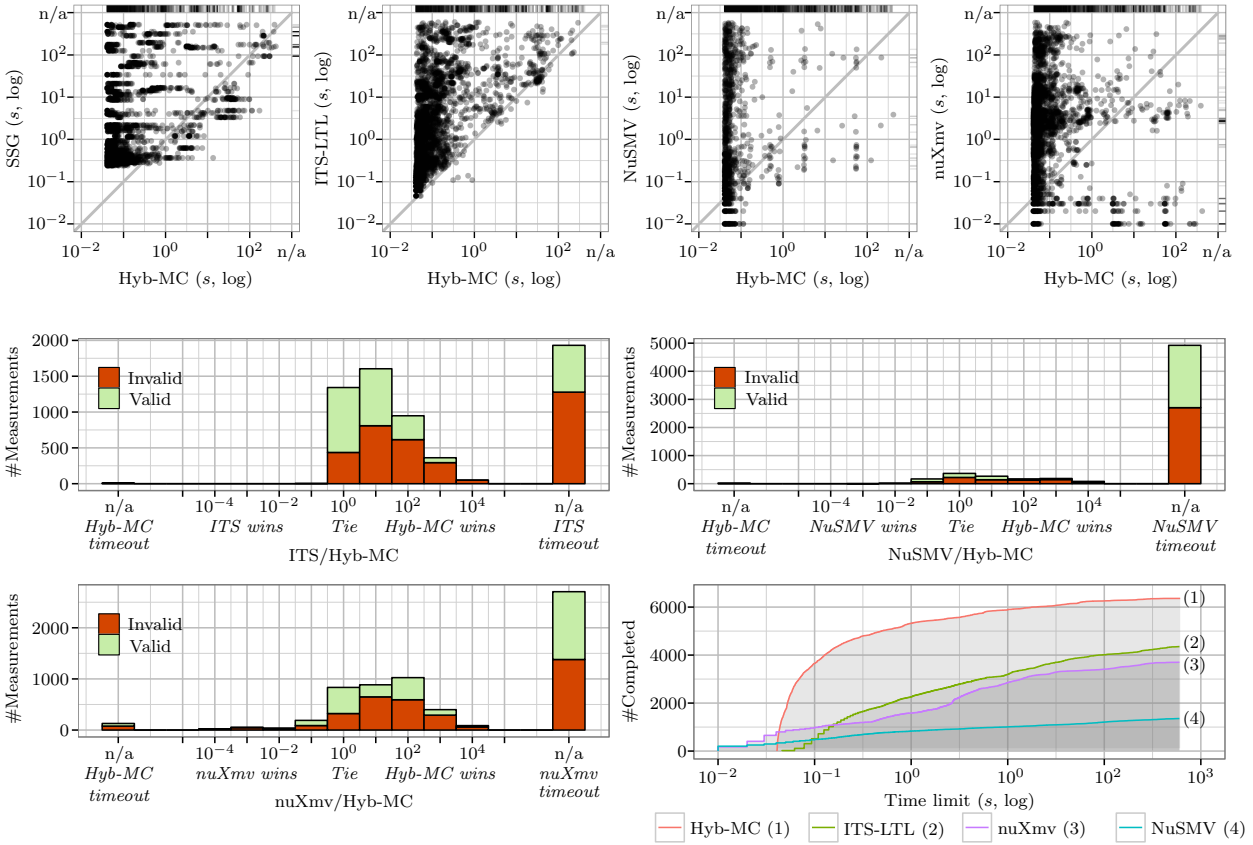


Fig. 8. Measurement results.

with saturation was also done for each model instance to prove that saturation is not the single reason of high performance—on-the-fly model checking can finish much earlier than a simple state space generation.

Results can be seen on the scatter plots of Figure 8. The four scatter plots show the comparison of Hyb-MC to simple state space generation (SSG) with saturation and the three chosen tools. Each dot represents a benchmark case. The horizontal and vertical axes measure the runtime of Hyb-MC and the other tool, respectively. A dot above the diagonal line is a benchmark case that Hyb-MC solved faster than the competitor. The borders of the diagrams represent cases where one of the tools did not finish under the time limit. Cases in which neither of the tools finished are not shown on this diagram.

As the top-left plot suggests, model checking with Hyb-MC is usually faster than simple state space generation. The main cause of this is the algorithm's on-the-fly operation and efficient incremental operation (i. e., a low degree of redundancy). There are also some cases where state space generation could be finished, but model checking was unsuccessful. These cases show that incremental operation and the presented optimizations cannot always compensate the overhead of model checking complex properties.

Comparing to the other model checking tools, the vast majority of cases show the competitiveness of the presented algorithm. Since the scales of the axes are logarithmic, the distance of a dot from the diagonal indicate an exponential difference in the runtimes of the tools.

This difference is visualized on the bar charts of Figure 8. The bar charts show the quantity t_A/t_B with a logarithmic horizontal axis, where t_A and t_B are the runtime of the compared algorithms. Bar charts of Figure 8 are colored to show the distribution of valid and invalid properties. Hyb-MC is better than the other tools more often in refuting an invalid property than in proving valid ones. This shows the efficiency of the SCC detection algorithm compared to the other approaches, but even valid cases show a decent speedup compared to the other three tools, justifying the product computation algorithm and the use of abstraction and recurring states as a means to prove the absence of counterexamples.

Table 1. SCC detection statistics in different groups of cases.

Cases	Time spent with SCC detection	Prevented symbolic runs	Prevented by	
			Recurring	Abstraction
All	16.9%	99.7%	89.7%	10.3%
Valid	13.5%	99.7%	89.0%	11.0%
Invalid	19.6%	99.7%	91.3%	8.7%
Obligation	13.7%	99.7%	86.3%	13.7%
Not obligation	23.2%	99.7%	93.0%	7.0%
Echo (2D)	2.7%	100.0%	100.0%	0.0%
Eratosthenes	65.9%	62.9%	19.0%	81.0%

The last diagram in Figure 8 shows how many of the cases finished with runtime under the value of the horizontal axis, i. e., how many cases would have finished if the timeout was set to a specific value. This diagram indirectly suggests the scalability of the algorithms. The initial delay of Hyb-MC and ITS-LTL is due to SPOT building the Büchi automaton from the LTL property.

It is interesting to note some special features of the tools. ITS-LTL performed the best among the chosen competitors, but it could outperform Hyb-MC in very few cases only. Since it is also based on saturation and uses abstraction to perform on-the-fly model checking, these results are the most significant among all the others.

Although NuSMV performed the worst of all competitors, it could beat Hyb-MC in more cases and also more significantly than ITS-LTL. NuSMV is the oldest of all the tools and is supposed to be better with synchronous models, so these cases deserve future attention.

Results of nuXmv are very different from results of the other tools. This is not surprising, since it uses a SAT-based algorithm with different techniques and strengths. It is also the one that outperformed Hyb-MC most of the times, it even managed to process many cases that caused a timeout in Hyb-MC. It is also interesting to note that nuXmv proved to be the only tool sensitive to the category of the checked property.

Analysis of runtime results measured on model families showed other interesting differences in the scalability of the tools. NuSMV and nuXmv performed significantly better on models where instances of the model were scaled in the domain of state variables. On the other hand, saturation-based tools Hyb-MC and ITS-LTL were much better on models that scaled in the number of variables (components).

7.3.2. Efficiency of the Presented Techniques

During the measurements, Hyb-MC spent only 17% of the time computing SCCs. Overall, 359 084 symbolic fixed point computations were started, while abstraction and explicit algorithms prevented $1.22 \cdot 10^8$ runs of symbolic SCC computation, 99.7% of all the cases. 90% of these cases were prevented by the absence of recurring states (as a first check), while the remaining 10% were the cases where explicit runs on node-wise abstractions managed to find even more evidence.

Table 1 shows the discussed statistics for some interesting subsets of the benchmark cases. Not surprisingly, Hyb-MC spends more time looking for SCCs in case of invalid properties. The efficiency of recurring states and explicit search slightly varies, but not significantly. The difference is greater between obligation and more complex formulae: complex properties (such as progress properties) require more time spent on SCC detection. The share of recurring states and explicit search also varies a bit more.

Echo (2-dimensional instances) and Eratosthenes are two extreme models in terms of time spent on SCC detection. Echo was one of the hardest models for Hyb-MC, only 26% of all the cases were solved in the 2-dimensional family, all of which were valid. The fact that symbolic SCC detection was never called in these cases suggests that Hyb-MC fails here on invalid cases, where SCC detection would be necessary (cases that timed out do not contribute to these statistics). Eratosthenes, on the other hand, had all of its cases solved, and there also were valid properties. In spite of this, almost two thirds of the time was spent on computing SCCs. Explicit search also shows extremely high efficiency here, although the percentage of prevented symbolic runs is the only value among the models that is under 90%.

8. Summary

LTL model checking of asynchronous systems is a computationally difficult problem. Various techniques and algorithms were developed in the history to tackle the state space explosion problem which is inherent in concurrent systems, and to combat the complexity of LTL model checking. We address this complex problem in our work by introducing a new, hybrid LTL model checking algorithm for asynchronous systems. The proposed approach combines the advantages of various algorithms in a novel way. Saturation explores the possible states of the system and constructs the synchronous product on the fly with the help of a synchronization method built on top of constrained saturation.

A new incremental fixed-point algorithm decomposes the model checking problem into smaller, component-wise queries and computes local model checking results. In order to decrease the number of symbolic fixed point computations, the paper introduces a scalable abstraction framework, which efficiently filters SCC computations by using local explicit model checking runs.

The paper presented the following new algorithms:

- An efficient on-the-fly synchronous product generation algorithm based on saturation;
- An incremental symbolic fixed point computation algorithm for SCC detection;
- An abstraction technique to support inductive reasoning on the absence of SCCs;
- A unique hybrid model checking algorithm combining explicit state traversal with symbolic state space representation.

As a theoretical result, the paper also contains the proofs for the correctness of the presented algorithms.

The new algorithms were implemented in the PetriDotNet framework and extensive measurements were conducted to examine efficiency. The tool was compared to industrial and academic model checking tools. Although the implementation is only in the prototype phase, it turned out to be quite competitive and could solve more of the benchmarks cases than any of its competitors.

The presented algorithm has a huge potential for future developments. Following the idea of driving the symbolic algorithm with explicit runs, a promising direction is to combine partial order reduction with symbolic model checking. In addition, advanced representations of the properties can also be used to further improve the speed of model checking.

Acknowledgments

This work was partially supported by the ARTEMIS JU and the Hungarian National Research, Development and Innovation Fund in the frame of the R5-COP project. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions.

References

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BSHZ11] Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In Per Bjesse and Anna Slobodová, editors, *Proc. of the International Conference on Formal Methods in Computer-Aided Design*, pages 144–153. FMCAD Inc, 2011.

[Büc62] J. Richard Büchi. On a decision method in restricted second order arithmetic. In Ernest Nagel, Patrick Suppes, and Alfred Tarski, editors, *Proc. of the 1960 International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford Univ. Press, Stanford, California, 1962.

[BZC99] Armin Biere, Yunshan Zhu, and Edmund M. Clarke. Multiple state and single state tableaux for combining local and global model checking. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 1999.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Alberto Griggio, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. Technical report, Fondazione Bruno Kessler, 2014.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim G. Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[CGH97] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[CLS01] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2001.

[CLY07] Gianfranco Ciardo, Gerald Lüttgen, and Andy J. Yu. Improving static variable orders via invariants. In Jetty Kleijn and Alex Yakovlev, editors, *Petri Nets and Other Models of Concurrency – ICATPN 2007*, volume 4546 of *Lecture Notes in Computer Science*, pages 83–103. Springer, 2007.

[CMCH96] Edmund M. Clarke, Kenneth L. McMillan, Sergio V. Campos, and Vasiliki Hartonas-Garmhausen. Symbolic model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422. Springer, 1996.

[CMS03] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. Saturation unbound. In Hubert Garavel Hubert.Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 379–393. Springer, 2003.

[CMS06] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. The saturation algorithm for symbolic state-space exploration. *International Journal on Software Tools for Technology Transfer*, 8(1):4–25, 2006.

[CVWY91] Costas A. Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory efficient algorithms for the verification of temporal properties. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242. Springer, 1991.

[DKPT11a] Alexandre Duret-Lutz, Kaïs Klai, Denis Poitrenaud, and Yann Thierry-Mieg. Combining explicit and symbolic approaches for better on-the-fly LTL model checking. *CoRR*, abs/1106.5700, 2011. `http://arxiv.org/abs/1106.5700`.

[DKPT11b] Alexandre Duret-Lutz, Kaïs Klai, Denis Poitrenaud, and Yann Thierry-Mieg. Self-loop aggregation product – A new hybrid approach to on-the-fly LTL model checking. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, volume 6996 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2011.

[DP04] Alexandre Duret-Lutz and Denis Poitrenaud. SPOT: An extensible model checking library using transition-based generalized Büchi automata. In *Proc. of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 76–83, 2004.

[EC80] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, Secaucus, NJ, USA, 1996.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Piotr Dembinski and Marek Sredniawa, editors, *Proc. of the International Symposium on Protocol Specification, Testing and Verification*, pages 3–18. Chapman & Hall, Ltd., 1995.

[HIK04] Serge Haddad, Jean-Michel Ilié, and Kaïs Klai. Design and evaluation of a symbolic and abstraction-based model checker. In Farn Wang, editor, *Automated Technology for Verification and Analysis*, volume 3299 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 2004.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70. ACM, 2002.

[HKK+09] Lom M. Hillah, Ekkart Kindler, Fabrice Kordon, Laure Petrucci, and Nicolas Treves. A primer on the Petri Net Markup Language and ISO/IEC 15909-2. *Petri Net Newsletter*, 76:9–28, 2009.

[HPY97] Gerard J. Holzmann, Doron Peled, and Mihalis Yannakakis. On nested depth first search. In Gerard J. Holzmann Jean-Charles Grégoire and Doron A. Peled, editors, *The Spin Verification System*, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 81–89. AMS, 1997.

[KP08] Kaïs Klai and Denis Poitrenaud. MC-SOG: An LTL model checker based on symbolic observation graphs. In Kees M. van Hee and Rüdiger Valk, editors, *Applications and Theory of Petri Nets*, volume 5062 of *Lecture Notes in Computer Science*, pages 288–306. Springer, 2008.

[Kri63] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[McM92] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992. UMI Order No. GAX92-24209.

[McM03] Kenneth L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt, Jr. and Fabio Somenzi, editors, *Lecture Notes in Computer Science*, volume 2725, pages 1–13, 2003.

[MD98] D. Michael Miller and Rolf Drechsler. Implementing a multiple-valued decision diagram package. In *Proc. of the 28th IEEE International Symposium on Multiple-Valued Logic*, pages 52–57, 1998.

[MDVB15] Vince Molnár, Dániel Darvas, András Vörös, and Tamás Bartha. Saturation-based incremental LTL model checking with inductive proofs. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 643–657. Springer, 2015.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, NY, USA, 1992.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[Pel98] Doron Peled. Ten years of partial order reduction. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer, 1998.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proc. of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[SBJ14] Marcin Szpyrka, Agnieszka Biernacka, and Biernacki Jerzy. Methods of translation of Petri nets to NuSMV language. In Louchka Popova-Zeugmann, editor, *Concurrency, Specification and Programming*, volume 1269 of *CEUR Workshop Proceedings*, pages 245–256, 2014.

[SC06] Radu I. Siminiceanu and Gianfranco Ciardo. New metrics for static variable ordering in decision diagrams. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2006.

[SRB02] Fabio Somenzi, Kavita Ravi, and Roderick Bloem. Analysis of symbolic SCC hull algorithms. In Mark D. Aagaard and John W. O'Leary, editors, *Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2002.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[STV05] Roberto Sebastiani, Stefano Tonetta, and Moshe Y. Vardi. Symbolic systems, explicit properties: On hybrid approaches for LTL symbolic model checking. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 350–363. Springer, 2005.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[TMIP04] Yann Thierry-Mieg, Jean-Michel Ilié, and Denis Poitrenaud. A symbolic symbolic state space representation. In David de Frutos-Escrig and Manuel Núñez, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2004*, volume 3235 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2004.

[Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham Birtwistle, editors, *Logics for Concurrency*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, 1996.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of the Symposium on Logic in Computer Science*, pages 332–344. IEEE Computer Society, 1986.

[WBH+06] Chao Wang, Roderick Bloem, Gary D. Hachtel, Kavita Ravi, and Fabio Somenzi. Compositional SCC analysis for language emptiness. *Formal Methods in System Design*, 28(1):5–36, 2006.

[ZC09] Yang Zhao and Gianfranco Ciardo. Symbolic CTL model checking of asynchronous systems using constrained saturation. In Zhiming Liu and Anders P. Ravn, editors, *Automated Technology for Verification and Analysis*, volume 5799 of *Lecture Notes in Computer Science*, pages 368–381. Springer, 2009.

[ZC11] Yang Zhao and Gianfranco Ciardo. Symbolic computation of strongly connected components and fair cycles using saturation. *Innovations in Systems and Software Engineering*, 7(2):141–150, 2011.